(1)

TAQL: A Problem Space Tool for Expert System Development

Gregg R. Yost
May 1, 1992
CMU-CS-92-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Submitted to Carnegie Mellon in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. The research was also supported in part by Digital Equipment Corporation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Digital Equipment Corporation or the U.S. Government.

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

TAQL: A Problem Space Tool for Expert System Development

GREGG RICHARD YOST

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Statement A per telecon Chahira Hopper
WL/AAAT
WPAFB,OH 45433

NWW 8/10/92

ACCEPTED:

Allen Newell — **MAJOR PROFESSOR** — 24 Apr 92 — **DATE**

[illegible] — **DEAN** — 5/1/92 — **DATE**

APPROVED:

[illegible] — **PROVOST** — 1 June 1992 — **DATE**

Abstract

In recent years, the method-based approach to knowledge acquisition has received a great deal of attention. The method-based approach abandons traditional AI programming techniques and languages. Instead, method-based knowledge acquisition tools attempt to interact with the task developer in domain terms rather than computational terms. These tools are built around highly-structured problem-solving methods, and use knowledge of the built-in methods to drive knowledge acquisition. Method-based tools can be quite powerful, but the highly-structured method built into each tool limits its scope to a narrow range of tasks. Furthermore, there are some expert system tasks for which no method-based tool has yet been built, despite serious efforts to do so. SALT, KNACK, and OPAL are well-known method-based knowledge acquisition tools.

Method-based tools are designed by identifying five basic tool components:

1. A *task type* that refers to the class of applications that the tool can be used to implement.
2. A *method* that can perform tasks of this type. The method has a number of *knowledge roles* that must be filled with domain knowledge to form a complete expert system.
3. A *computer representation* for the method and the knowledge that fills its roles.
4. A *task language* in which the task developer describes the knowledge that will fill the method's roles.
5. A *mapping* from the task language to the method's knowledge roles and ultimately to the computer representation of those roles.

My thesis is that this same design strategy can be applied profitably to much broader, weaker methods than those that have been used in traditional method-based tools. I demonstrate my thesis by applying this design strategy to the problem-solving method that underlies Soar, a general-purpose intelligent architecture. The resulting tool, TAQL, generally cannot be used by domain experts — using it requires programming skills. However, in a sequence of task-development experiments, TAQL outperformed both SALT and KNACK, and proved to greatly surpass existing method-based tools in scope. The experiments also indicate that TAQL can be learned quickly and that it scales well to large expert-system tasks.

Acknowledgements

After all the years of research and experimentation and the countless hours spent turning fuzzy ideas into science, it comes as a bit of a surprise that this last section I am writing is one of the most difficult. How, in one page, can I possibly acknowledge all of the forces that have shaped my academic career? I surely cannot even be aware of all of those forces. But I will do my best.

I must start with my advisor, Allen Newell. He has not simply guided my research; he has shown me what it means to be a scientist. What were dead ends in my eyes were to Allen just signposts along a research path, albeit a winding one. In the way that only a great teacher can, Allen has educated me without ever making me feel that I was being instructed.

I have benefited from John McDermott's guidance even since he offered me my first programming job when I was an undergraduate. Our discussions of my research have been very productive, and he has always tried to make sure that I don't lose sight of real-world constraints.

Thanks also to my other two committee members, Mark Fox and Art Westerberg, for their input on my research.

Erik Altmann and Bob Doorenbos helped design, implement and maintain TAQL. They also spent considerable time helping to run the TAQL experiments and reading and commenting on drafts of my dissertation. I cannot thank them enough for taking time away from their own research to assist in the TAQL effort.

The Carnegie Mellon Computer Science Department has been a wonderful and stimulating environment to work in. I am particularly indebted to the Soar group — not just at CMU, but to its members worldwide. A special thanks goes to all of the people who have used TAQL for building their own systems. Their feedback has been invaluable.

My friends have kept me not only sane but happy. Special thanks go to my very best friends and companions: Catherine, Beth, Cristina, Dave, Don, Paul, Rachel, Renee, Scott, Scott, and Sid.

And, finally, I acknowledge the debt I owe to my parents. From my earliest years they taught me to value education. I could not have persevered through twenty-four years of schooling without their support and the values they so unobtrusively imparted. Mom and dad, I dedicate this dissertation to you.

To my parents

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

In recent years, an approach to knowledge acquisition exemplified by McDermott's role-limiting methods (McDermott, 1988) and Chandrasekaran's generic tasks (Chandrasekaran, 1986) has become increasingly popular. I will refer to this approach as *method-based knowledge acquisition*, since it uses information about the structure of a task-specific problem-solving method to drive knowledge acquisition.

The method-based approach abandons traditional AI programming techniques and languages. Instead, method-based knowledge acquisition tools attempt to interact with the task developer at the *knowledge level* (Newell, 1982). Ideally, the need for a highly-trained programmer is eliminated, since the domain expert can interact directly with the tool using the concepts of the domain. Method-based tools include SALT (Marcus et al., 1985; Marcus, 1988), KNACK (Klinker, 1988), MOLE (Eshelman et al., 1988), and OPAL (Musen et al., 1988).

To illustrate, I will briefly describe three well-known method-based tools. Then I will present the general structure that method-based tools have in common.

1.1. SALT

SALT (Marcus et al., 1985; Marcus, 1988) is a tool for building expert systems that construct a design subject to various constraints. For example, it has been used to build VT (Marcus, Stout & McDermott, 1988), an expert system that designs an elevator system meeting functional and safety constraints. SALT is built around a method called *propose-and-revise*. This method constructs a design incrementally. It proposes a solution for one aspect of the design at a time. If the partial design ever violates any constraints, the method revises prior decisions to remedy the problem. The method continues until it either finds a complete design that meets all constraints, or determines that there is no such design.

Propose-and-revise forms the skeleton of a SALT-built expert system. The method has three *knowledge roles* that must be filled with domain knowledge to form a complete expert system. These roles are to propose a design extension, identify a constraint violation, and fix a constraint violation. SALT interacts with the domain expert to acquire the domain knowledge that fills these three roles. Knowledge is entered by filling out role-specific schemas. For example, the following filled-out SALT schema is taken from the elevator-configuration domain. It is for the *identify a constraint violation* role, and specifies that a minimum of 6.5 inches of door space is required for two-speed elevator doors. The aspects on the left are pre-defined by SALT. The values on the right are supplied by the domain expert.

1	**Constrained value:**	**DOOR-SPACE**
2	**Constraint type:**	**MINIMUM**
3	**Constraint name:**	**MINIMUM-DOOR-SPACE**
4	**Precondition:**	**DOOR-SPEED = TWO**
5	**Procedure:**	**CALCULATION**
6	**Formula:**	**6.5**
7	**Justification:**	**NONE**

Finally, SALT maps the knowledge it has acquired for the roles into the representations it uses in the final implementation (an OPS5 production system (Forgy, 1981; Brownston, Farrell, Kant, and Martin, 1985)).

The hope behind SALT is that domain experts for the kinds of design problems SALT supports will find it natural to describe their knowledge in terms of design extensions, constraints, and fixes. If this is true, domain experts will be able to use SALT to build expert systems with little or no assistance from a knowledge engineer.

1.2. KNACK

KNACK (Klinker, 1988) creates expert systems that assist in reporting tasks. For example, one KNACK-built system creates reports that evaluate electro-mechanical designs with respect to their resistance to the electromagnetic pulses generated by nuclear blasts.

KNACK-built systems use a method called *acquire-and-present* to create a report. This method requires five kinds of knowledge:

- Knowledge that defines the skeletal structure of the report to be generated.
- Knowledge that defines the fragments of text that potentially comprise the report.
- Knowledge that identifies the information relevant to each part of the report.
- Knowledge about strategies that can be used to acquire a piece of information (example strategies are asking questions and inference from something already known).
- Knowledge that integrates newly-acquired information with existing information.

Acquire-and-present operates by attempting to construct a report from beginning to end, as defined by the skeletal report knowledge. For each part of the report, it identifies the information needed to complete that report section, and identifies and applies the appropriate strategies for acquiring that information. As new information is acquired, it is integrated with existing information as necessary. For example, when the system is informed of a new component in a physical design, it must determine how the new component connects to existing design components. Finally, acquire-and-present instantiates report fragments with the collected information and adds them to the report.

KNACK creates a reporting system by having the domain expert enter a sample report. It uses a variety of heuristics to analyze and generalize the report. In doing so, it builds models of both the domain and of the general report structure. To assess the accuracy of its models, it instantiates them in various ways and presents the results to the domain expert. If the domain expert finds the instantiated models inadequate, KNACK and the domain expert iteratively refine them. The knowledge KNACK collects in this phase supplies the five kinds of knowledge acquire-and-present needs to operate. Finally, KNACK maps the collected knowledge to an OPS5 production system that performs the reporting task.

1.3. OPAL

OPAL (Musen et al., 1988) lets medical specialists create cancer-treatment plans for use in the ONCOCIN (Shortliffe et al., 1981) expert system. ONCOCIN makes cancer therapy recommendations based on clinical information about the patient and a knowledge base of treatment protocols.

ONCOCIN uses skeletal plan refinement to build treatment plans. It starts with a very abstract plan and repeatedly selects and refines plan elements to create a progressively more detailed plan. Skeletal plan refinement in ONCOCIN requires three kinds of knowledge: knowledge of protocol components and their structural relationships; knowledge about how to instantiate treatment parameters (for example, drug dosages); and knowledge about how treatments progress over time. ONCOCIN represents these three kinds of knowledge using frames, production rules, and finite state machines, respectively.

Entering treatment protocols in terms of these knowledge representations proved unwieldy, and OPAL was built to overcome this problem. OPAL provides a graphic interface based on the domain's concepts. The interface includes forms for defining protocols and describing how to instantiate treatment parameters. It also provides a visual programming language for building diagrams of how treatments progress over time. A medical specialist fills out the forms and constructs the treatment diagrams.

OPAL translates this domain-oriented information into ONCOCIN's basic knowledge representations.

1.4. Method-based tool components

The brief descriptions of SALT, KNACK, and OPAL illustrate the basic components that comprise all method-based tools:

1. A *task type*. The task type refers to the class of applications that the tool can be used to implement.
2. A *method* that can perform tasks of this type. The method has a number of *knowledge roles* that must be filled with domain knowledge to form a complete expert system.
3. A *computer representation* for the method and the knowledge that fills its roles.
4. A *task language* in which the task developer describes the knowledge that will fill the method's roles.
5. A *mapping* from the task language to the method's knowledge roles and ultimately to the computer representation of those roles.

Method-based tools are usually intended to be used directly by a domain expert who may have no programming skills. To achieve this, the tool designer must carefully balance the five components listed above. The task language must be one the domain expert can understand. Also, the language and method must be limited enough in scope so that identifying and implementing the mapping functions is feasible. This in turn limits the types of tasks to which each tool can be applied. Thus, while a given method-based tool may be quite powerful for appropriate tasks, the range of appropriate tasks is very narrow.

1.5. Advantages and disadvantages of method-based tools

Method-based tools have a number of advantages. They are usually designed to interact with the user at the task level. Therefore, domain experts can often use these tools directly without the assistance of a knowledge engineer. Furthermore, the tool itself performs the mapping from domain knowledge to computational terms. The tool can use this information to produce understandable traces and explanations, which should ease both application development and application maintenance. Also, since the tool has an explicit model of the roles played by each kind of knowledge in the system, it can perform sophisticated knowledge-level consistency and completeness tests.

Method-based tools also have a number of disadvantages. These tools obtain their power at the expense of generality. Each tool uses a problem-solving method and external language that is appropriate for only a very narrow range of tasks. Often, even slight variants of a task that a tool is well-suited for cannot be expressed within the tool. In practice, when this happens, a knowledge engineer who is familiar with the tool's internal representations steps in to hand-code parts of the system (see, for example, (Klinker et al., 1989)). Unfortunately, as soon as this is done one loses the primary benefits of method-based tools: the system's knowledge is no longer entirely in domain terms, and the hand-coded parts do not correspond to any of the method's knowledge roles. Section 6.1.2 describes this brittleness problem in more detail.

If for each potential expert system task there existed a method-based tool capable of encoding that task, and it was easy to select an appropriate tool for a given task, then the narrow scope of individual method-based tools would not be such a serious problem. Unfortunately, this is far from true. Existing method-based tools span only a small fraction of potential tasks. Further, there are tasks for which no appropriate method-based tool has been constructed, despite serious efforts (for example, the computer configuration task performed by the XCON expert system (Barker et al., 1989) — see (McDermott, 1988)). There is also no reliable way to tell in advance whether a given method-based tool will be adequate for a particular task. Currently, several research groups are exploring ways to build method-based tools in which method components can be assembled and recombined in novel ways to better suit a task's requirements (Klinker et al., 1990; Puerta et al., 1991). To date, however, these tools have not substantially expanded the scope of method-based tools.

1.6. Statement of thesis

Method-based tools are designed by identifying the five basic tool components listed in Section 1.4. Traditional method-based tools have applied this design strategy only to strong, narrow methods. My thesis is that this same design strategy can be applied profitably to much broader, weaker methods. Doing so involves changing some of the assumptions of traditional method-based research — in particular, using the tool will require programming skills. However, even for a weak method one can identify the five central tool components listed above, and use these components to develop a highly-effective expert system development tool. I demonstrate my thesis by applying this design strategy to the problem-solving method that underlies Soar (Rosenbloom, Laird, Newell, and McCarl, 1991; Laird, Congdon, Altmann and Swedlow, 1990), a general-purpose intelligent architecture. The resulting tool, TAQL, has much broader scope than current method-based tools and yet outperforms them in a sequence of experiments. Furthermore, TAQL's use of a weak, broadly-applicable method allows it to overcome the disadvantages of method-based tools that were listed in the previous section.

1.7. A reader's guide

Chapter 2 gives the details of how the strategies used to design method-based tools apply to Soar. It presents Soar's problem-solving method, PSCM, and a corresponding computer language, TAQL. Chapter 2 also describes the external task-description language and how it maps to PSCM. It concludes with a discussion of how the design strategies applied to Soar differ from the traditional ones, and a discussion of how the strong methods used in traditional tools can be embedded in PSCM.

Chapter 3 presents the evaluation criteria and methodology that were used to evaluate TAQL, my expert system development tool. The methodology centers around a series of task-development experiments used to evaluate TAQL to guide TAQL's development. The primary evaluation criteria are *scope* and *effectiveness*. *Scope* refers to the range of tasks TAQL can be used to develop. *Effectiveness* refers to how easily a programmer can implement a task using TAQL.

Chapter 4 describes the TAQL task-development experiments. It describes the tasks, subjects, and versions of TAQL used in the experiments, and presents the data collected. It also describes how and why TAQL evolved between experiment rounds in response to the experiments' results.

Chapter 5 evaluates TAQL's scope and effectiveness based on the experimental data. The data indicates that TAQL has a very broad scope and outperforms both SALT and KNACK. It also indicates that novice TAQL programmers can quickly reach expert-level performance and that TAQL is equally effective for small and large tasks.

Chapter 6 summarizes the results and the main contributions of the thesis. It discusses the aspects that led to TAQL's success, which exceeded initial expectations. Briefly, I attribute TAQL's success to

1. The principled way in which TAQL was developed: namely, by using the same strategies that have been used to build traditional method-based tools.
2. The simplicity and flexibility of Soar's computational model (PSCM).

Appendix A tells how to obtain TAQL, Soar, and related documents. Appendix B includes a complete TAQL program, and Appendix C includes a complete domain-oriented task description for one of the small expert-system tasks used in the TAQL experiments. The log file for one of the experiment trials appears in Appendix D. Appendix E presents additional experiment data not presented in Chapter 4.

For readers interested in a less exhaustive presentation of the work, the following sections cover the most important topics and results: Chapter 1, Chapter 2 through Section 2.3, Chapter 3, Chapter 4 through Section 4.3, Chapter 5, and Chapter 6 through Section 6.2. A brief overview of the major results appears in (Yost, 1992a).

Chapter 2
Applying Method-Based-Tool Design Strategies to Soar

Traditionally, method-based tools have targeted highly-structured but narrow methods. As pointed out in Section 1.4, given the current state of the art, such limited methods are required in a tool that a non-programming domain expert will use. This chapter applies the strategies that have been used to design method-based tools to the much weaker method that underlies Soar. Consequently, the resulting tool is not suitable for use by a non-programmer. However, as shown in Section 5.2, the resulting tool can be more effective than traditional method-based tools and is applicable to a much wider range of tasks.

Constructing a method-based knowledge acquisition tool involves identifying the five central components listed in Section 1.4. For my Soar-based tool (TAQL), these five components are:

1. The *task type*: Soar is a general-purpose architecture that is intended to display a full range of intelligent behavior, from routine action to complex problem-solving. Thus, TAQL's task type comprises this same broad range of tasks.

2. The *method* that can perform tasks of this type: PSCM (*Problem Space Computational Model*), which is Soar's underlying problem-solving method. Section 2.1 describes PSCM and its knowledge roles.

3. The *computer representation* for PSCM and for the knowledge that fills its roles is a language called TAQL (*Task AcQuisition Language*), described in Section 2.2.

4. The *task language*: Natural language. Natural language was chosen as the task language because it is the language people commonly use to describe the range of tasks covered by the above task type. Section 2.3 discusses the nature and appropriateness of natural language task descriptions.

5. The *mapping* from domain-oriented natural-language task descriptions to PSCM's knowledge roles and finally to TAQL is described in Section 2.4.

TAQL together with its programming environment comprises my expert-system development tool. I refer to the complete tool simply as TAQL.

2.1. PSCM

PSCM (*Problem Space Computational Model*) (Newell, et al., 1991) is Soar's underlying problem-solving method. It defines the knowledge-level structures that the Soar architecture implements. PSCM's basic components are tasks, problem spaces, states, and operators. Tasks are particular problems to be solved. States consist of the objects relevant to the task. Operators manipulate and transform states. Problem spaces comprise states and operators and the knowledge that relates to them.

Tasks are performed in problem spaces. When a task is created, problem solving proceeds as follows:

```
Propose problem spaces to perform the task and select one
Propose initial states for the problem space and select one
While the current state is not a desired state:
    Propose operators to apply to the current state and select one
    Apply the operator to the current state, producing a new current
      state.
```

PSCM's knowledge roles are apparent from this method: propose and select problem spaces, initial states, and operators; apply operators; and recognize desired states. At each step in the method, PSCM attempts to complete the step by bringing to bear all available knowledge for the appropriate role. Sometimes, however, the available knowledge is not sufficient to specify a unique decision. For example, there might not be any relevant available knowledge, or the available knowledge may be conflicting. In such a case, PSCM automatically creates a new subtask to gather sufficient knowledge to make a decision. The above method recursively applies to the new subtask. Thus, a task in PSCM is performed not by a single space but by a cooperating set of problem spaces.

PSCM has one additional knowledge role that is not apparent from the method description: *state entailment*. A state entailment specifies that when some collection of structures is part of the state, then some other collection of structures is also part of the state. Operators define the core structure of a state. Entailments flesh out this structure with additional information. If an active entailment's conditions become unsatisfied, the structures it placed on the state are removed. This provides a simple form of belief maintenance.

PSCM also incorporates learning. A subtask completes when its subspace has gathered enough knowledge to permit further progress in the superspace. All of the gathered knowledge that contributes to the decision on how to proceed becomes part of the immediately-available knowledge in the superspace (some of the gathered knowledge may be irrelevant to the decision — such knowledge is not transferred to the superspace). Thus, on encountering similar situations in the future, PSCM will be able to proceed without generating the subtask.

2.1.1. An example

I will illustrate PSCM by developing problem spaces to perform an extremely simple task: toggle a light between off and on. This example is completely elementary, but shows in detail many of the PSCM knowledge roles. It does not illustrate search control, however, because each problem space will have only one operator.

Two problem spaces implement this task: *light*, the problem space that will toggle the light; and the *top-space* space, which sets up the task context. The top-space space contains an operator named *light-task*, which is applied by the light space. Figure 2-1 shows how these two problem spaces fit together and indicates the operators that are part of each space.

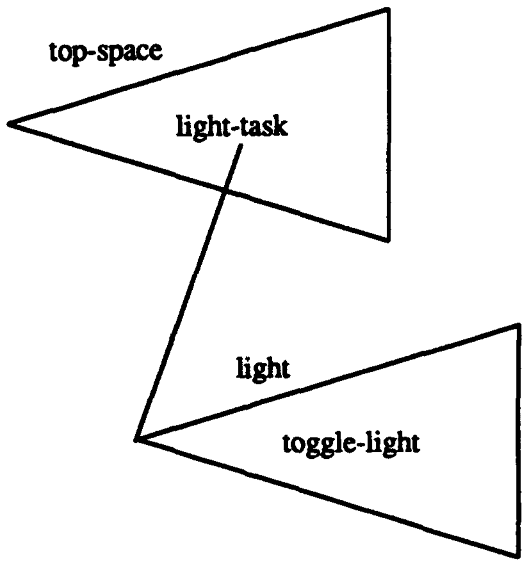


Figure 2-1: The problem spaces and operators for the light-toggling example

2.1.1.1. The top-space space

The top-space problem space exists solely to contain the light object that is to be toggled and to invoke the light-task operator. It is defined by defining the knowledge that fills each of its PSCM knowledge roles, as follows:

- *Problem space proposal*: None. This problem space is at the top of the problem-solving hierarchy, and it is selected automatically when problem-solving begins. Therefore no proposal knowledge needs to be defined.
- *Problem space selection*: None. The top space is automatically selected when problem-solving begins.
- *Initial state proposal*: The initial state in the top-space space contains the light object to be toggled.

- *Initial state selection*: None. Only one initial state is proposed.
- *Desired state recognition*: The goal test in the top space is satisfied when the light-task operator has completed.
- *State entailment*: None.

The top-space space has one operator:

- *Light-task*: This operator is proposed in the initial state. It is applied in the light space.

There is no knowledge in the operator-selection role for this space, because only one operator instance is ever proposed in it.

2.1.1.2. The light space

The light space applies the light-task operator. It toggles the status of the light object on the top state. Its PSCM knowledge roles are defined as follows:

- *Problem space proposal*: The light space is proposed whenever the light-task operator is selected in a problem space that lacks the operator-application knowledge necessary to produce the result.
- *Problem space selection*: None. Only one space, the light space, is proposed to apply the light-task operator.
- *Initial state proposal*: The initial state in the light space contains a copy of the light object from the top state. Only one aspect of the light is represented: it on/off status.
- *Initial state selection*: None. Only one initial state is proposed.
- *Desired state recognition*: The goal test in this space is satisfied when the copy of the light object on the local state has a status different from that of the light object on the top state.
- *State entailment*: None.

The light space has one operator. The knowledge that fills its operator-proposal and operator-application roles is defined below.

- *Toggle-light*: An instance of the toggle-light operator is proposed whenever the state contains a light object. It takes a single parameter: the light object to toggle. The toggle-light operator is applied by changing the light status to off if it is on, and to on if it is off.

There is no knowledge in the operator-selection role for this space, because only one operator instance is ever proposed in it.

This section has described a set of problem spaces using a mixture of problem-space terminology and knowledge-level terminology. It divides the knowledge needed to perform the task into problem spaces and their knowledge roles. The knowledge content of each role, however, is specified at the knowledge level. For example, the descriptions specify the effects each operator must have, but do not specify the computational mechanisms and representations that will achieve those effects. The next section describes TAQL, a computer language that expresses PSCM computations. Section 2.2.1 presents TAQL code for the problem spaces described here. The TAQL code defines all of the computational details that are left undefined in the PSCM problem-space descriptions. The mixed knowledge-level/symbol-level description of problem spaces used here is a useful intermediate level for designing TAQL systems.

2.2. TAQL

PSCM provides a conceptual structure for describing tasks. To build executable systems in PSCM, one needs a formal computer language that corresponds to PSCM. Soar's production language is one such language. However, Soar productions do not correspond in an obvious way to PSCM concepts. The productions have a very uniform structure with no syntactic differentiation with respect to problem-space concepts. This section describes TAQL (*Task AcQuisition Language*) (Yost and Altmann, 1991a; Yost and Altmann, 1991b; Yost and Altmann, 1991c), a programming language with a much clearer correspondence to PSCM and its knowledge roles.

TAQL directly reflects the structure of PSCM. Thus, a TAQL specification consists of a set of *TAQL constructs*, called *TCs*, each of which describes some aspect of a PSCM knowledge role. A Common Lisp program compiles TCs into Soar productions. When loaded into Soar along with a set of runtime support productions, these productions implement the task the TCs describe. This compilation is fully automated and efficient: it does not take much longer to load a file of TCs than it does to load the productions those TCs generate. Because of the close correspondence between PSCM and TAQL, mapping from one to the other is straightforward. Section 2.2.1 illustrates this, by presenting the TAQL code that corresponds to the problem spaces for the light-toggling task described in Section 2.1.1. Section 2.4.1 also illustrates the mapping from PSCM to TAQL.

Each TC is a list consisting of the TC type and a name for the TC instance, followed by a list of arguments. Each argument specifies some aspect of the TC's PSCM knowledge role. For example, in terms of PSCM, the aspects that must be defined for the operator-proposal role are the problem space it applies in, the operator object to be proposed, and the conditions under which the operator should be proposed. The *propose-operator* TC directly specifies these aspects as the values of the *:space*, *:op*, and *:when* keywords, respectively. The set of keywords for each TC is defined by TAQL, not by the programmer; the number of keywords for each TC is relatively small, usually five or fewer. Data is represented in TAQL using attribute/value structures of the form used in

Soar. The following propose-operator TC is from the familiar monkey-and-bananas task. It proposes that the monkey reach for the bananas whenever it is at the bananas.

```
(propose-operator monkey-and-bananas*po*reach
  :space monkey-and-bananas
  :when ((state ^monkey-at bananas))
  :op reach)
```

Table 2-1 lists all of the TAQL constructs, grouped by their associated PSCM knowledge role. The relation of most of the TCs to their PSCM role is obvious from the TC's name, but a few deserve further comment. The *evaluate-object* and *evaluation-properties* TCs define how to evaluate alternative objects (for example, operators) when the immediately-available selection knowledge doesn't uniquely determine a choice. These TCs are listed under the operator-application role because although they ultimately yield selection knowledge, they do so by specifying how to apply an evaluation operator that is part of Soar's default knowledge. See Section 4.4.2.1 for more details on these TCs.

The *propose-superobjects* TC returns knowledge from a problem space whose subtask

Knowledge Role	TAQL Construct (TC)
Problem-space proposal	propose-space propose-superobjects
Initial-state proposal	propose-task-state propose-initial-state propose-superobjects
Operator proposal	propose-task-operator propose-operator propose-superobjects
Operator application	apply-operator result-superstate evaluate-object evaluation-properties
Desired-state recognition	goal-test-group
Selection	prefer compare operator-control
Entailment	augment

Table 2-1: PSCM roles and their associated TCs

is to propose a problem space, initial state, or operator. The *result-superstate* TC returns the result state for an operator that is being applied in a subtask.

The TAQL constructs define TAQL's core structure. TAQL also has a powerful programming environment, including facilities for viewing a program abstractly and for declaring the general structure of problem spaces, operators, and data objects. Section 4.4 describes several of these facilities in more detail. The TAQL manual (Yost and Altmann, 1991a) and the TAQL 3.1.4 release notes (Yost and Altmann, 1991b) provide complete details. The TAQL tutorial (Yost and Altmann, 1991c) is a better introduction to TAQL than the manual, which is written as a reference manual. The tutorial contains several example TAQL programs. It does not, however, discuss many of TAQL's more advanced features.

2.2.1. An example

Section 2.1.1 presented PSCM definitions of two problem spaces. Together, these problem spaces implemented a very simple task: toggling a light between on and off. This section presents the TAQL code that implements those problem spaces. Appendix B includes a complete listing of the commented TAQL code for these spaces. As you are reading through this section, refer to the corresponding PSCM-level descriptions in Section 2.1.1 and note the close correspondence between PSCM concepts and TAQL code. This makes writing a TAQL program fairly straightforward once an abstract problem-space design has been completed.

2.2.1.1. The top-space space

TAQL predefines most of what happens in the top-space space. It automatically selects top-space as the initial space when a program is run, and gives it an empty initial state by default. The TAQL programmer typically specifies only a single operator to apply, called the *task operator*. The programmer defines the bulk of the task in the space that applies the task operator. When the task operator completes, TAQL automatically halts.

In this example, the task operator is the *light-task* operator, described in Section 2.1.1. The propose-task-operator TC proposes the task operator. It has one keyword, *:op*, that specifies which operator to use as the task operator. In this example, the TC is

```
(propose-task-operator top*pto*light-task
  :op light-task)
```

This example does not use the default empty initial state. Instead, the top state contains the signal-light object to be toggled, as specified in Section 2.1.1.1. The light object is represented as the value of a state attribute named *signal-light*; the light object has one attribute, *status*, which takes the values *on* or *off*. The following TC defines a top state where the light is initially off (TAQL uses Soar's attribute/value notation to describe objects).

```
(propose-task-state top*pts
  :new (signal-light ((light ^status off))))
```

These two TCs define all of the PSCM knowledge for the top space listed in Section 2.1.1.1 that is not predefined by TAQL.

2.2.1.2. The light space

The light space (Section 2.1.1.2) applies the light-task operator. It makes a local copy of the light object from the top state and applies the toggle-light operator to it. Once the light has been toggled, the task is done and the light space exits.

First, the propose-space TC proposes the light space. The *:function* keyword indicates that the space should be proposed when TAQL needs a space to apply the light-task operator.

```
(propose-space light*ps
  :function (apply operator light-task)
  :space light)
```

Next, the propose-initial-state TC defines the initial state. The *:copy* keyword with the *:copy-new* option specifies to make a duplicate copy of the value of the superstate's signal-light attribute, and place the copy on the initial state in the light space. TAQL provides several other keywords for creating new structures on the initial state.

```
(propose-initial-state light*pis
  :space light
  :copy (:copy-new (signal-light) :from superstate))
```

The desired states in the space are defined using the goal-test-group TC. Here, the goal-test-group TC checks that the status of the local copy of the light is different from the status of the light on the superstate.

```
(goal-test-group light*gtg*status-changed
  :space light
  :group-type success

  :when ((state ^signal-light <local-light>)
         (light <local-light> ^status <status>)
         (superstate ^signal-light <top-light>)
         (light <top-light> ^status <> <status>)))
```

When TAQL detects a desired state, it selects an operator named *final-state*. The result-superstate TC defines how to apply the final-state operator. It is very similar to the apply-operator TC that will be illustrated later, but is specialized for the final-state operator. Typically, a result-superstate TC places the results of applying the superoperator on the superstate. Here, the local copy of the light object (which has by now been toggled) is copied to the superstate, replacing the old light object there.

```
(result-superstate light*rs
  :space light
  :group-type success

  (edit :what superstate
        :copy (signal-light :remove target)))
```

Lastly, I define the operators in the space. The propose-operator TC defines when the operators are proposed, and the apply-operator TC defines how they are applied.

An instance of the toggle-light operator is proposed whenever there is a light object on the state. The operator takes one parameter, the light object to toggle. Here, the parameter is represented by a *light* attribute on the operator object that is proposed.

```
(propose-operator light*po*toggle-light
  :space light
  :when ((state ^signal-light <light>))
  :op (toggle-light ^light <light>))
```

Simple operator applications are defined using the apply-operator TC. For more complicated operators, a TAQL programmer typically defines a separate problem space in which the operator is applied. (In fact, the light space is such a space. It applies the light-task operator.) The toggle-light operator is very simple, so it is defined by an apply-operator TC. It edits the light object that is the operator's parameter, replacing the old value of its *status* attribute with the appropriate new value.

```
(apply-operator light*ao*toggle-light
  :space light
  :op (toggle-light ^light <light>)

  (edit :what (:none light <light>)
     :replace
        (status :by off
           :when ((light <light> ^status on)))
     :replace
        (status :by on
           :when ((light <light> ^status off)))))
```

2.2.1.3. Running the example code

This section shows how to run a TAQL program, for concreteness. Readers not interested in this level of detail may skip this section. In what follows, items typed by the user are underlined.

First, start Soar and load the TAQL compiler and runtime system into it.

```
[styraco!gry] Soar5
Allegro CL 3.1.12.2 [DECstation] (11/19/90)
Copyright (C) 1985-1990, Franz Inc., Berkeley, CA, USA

Soar (Version 5, Release 2)
Created August 26th, 1991
Bugs and questions should be sent to Soar-bugs@cs.cmu.edu
The current bug-list may be obtained by sending mail to
Soar-bugs@cs.cmu.edu with the Subject: line "bug list"
This software is in the public domain.

This software is made available AS IS, and Carnegie Mellon
University and the University of Michigan make no warranty
about the software or its performance.

See (soarnews) for news.
; Loading /usr0/gry/.soar-init.lisp.
; Loading /usr/misc/.Soar5/bin/Soar5.latest.patches.lisp.

<cl> (load "/usr/misc/.Soar5/lib/taql/3.1.4/load.lisp")

; Loading /usr/misc/.Soar5/lib/taql/3.1.4/load.lisp.

Disabling selected default productions: #*#*#*#*#*#*#**#*#*#**
#*#*
Loading TAQL support productions: *******************************
***********************************************************

TAQL 3.1.4
Created July 15, 1991

Bug reports should be sent to Soar-bugs@cs.cmu.edu.
Send comments on TAQL to Gregg.Yost@cs.cmu.edu or Erik.Altmann@c
s.cmu.edu.

t
```

Next, load the TAQL source code for this example. It is in a single file named light.taql (which appears in its entirety in Appendix B). Soar prints an asterisk for each production it loads. The TAQL compiler created these Soar productions at load time from the TAQL code.

```
<cl> (load "light.taql")

; Loading light.taql.
************
t
```

Now run TAQL for two decisions. At each decision, Soar chooses a goal, problem space, state, or operator (labeled *G:*, *P:*, *S:*, and *O:* in the traces below). The symbol after the label (for example, *g1*, *p4*, *s19*) is a unique Soar-generated object identifier. Indentation represents subgoaling. Goal objects are generated internally by Soar. Other objects are generated by the user's program.

```
<cl> (d 2)

0   G: g1
1   P: p4 (top-space)
2   S: s19
```

By this point, Soar has installed the top space and state. We now examine the contents of the top state, using Soar's *spo* (*Soar Print Object*) command. The second argument to spo is the maximum depth to print in an object that has substructure. The top state initially contains just the light object.

```
<cl> (spo s19 20)

(state s19 ^signal-light 114)
   (light 114 ^status off)
```

Now continue running the program until it halts. It goes into the light space, sets up the initial state, applies the toggle-light operator to the local copy of the light, returns the toggled light to the top space, and halts.

```
<cl> (run)

3   O: o15 (light-task)
4   ==>G: g22 (operator no-change)
5      P: p29 (light)
6      S: s36
7      O: o39 (toggle-light)
8      O: o34 (final-state)
       Space light succeeded in goal g22.
9   O: o6 (halt)
Applied task operator o15 (light-task).  Final state is s19.
End -- Explicit Halt
nil
```

Finally, again examine the top state. It now contains the toggled light object that was passed back from the light space. The *text-environment* and *dummy-att** attributes on the state were created automatically by the Soar and TAQL runtime systems, respectively. They are not relevant to this example.

```
<cl> (spo s19)

(state s19 ^text-environment t21 ^dummy-att* true
           ^signal-light n38)
nil
<cl> (spo n38)

(light n38 ^status on)
nil
```

2.3. Domain-oriented task descriptions

One of the five central knowledge acquisition tool components listed in Section 1.4 is the *task language* that the domain expert uses to describe domain knowledge. Method-based tools typically provide a language of terms and concepts tailored to the limited range of tasks the tool is targeted at. For example, SALT's (see Section 1.1) task language consists of statements about constraints, fixes, design extensions, and other concepts relevant to its built-in problem-solving method. Each kind of knowledge has a corresponding schema that must be filled out to define a piece of knowledge. The following is a filled-out SALT schema that defines one possible way to fix a particular constraint violation in the elevator-configuration domain:

1	**Violated constraint:**	**MAXIMUM-MACHINE-GROOVE-PRESSURE**
2	**Value to change:**	**HOIST-CABLE-QUANTITY**
3	**Change type:**	**INCREASE**
4	**Step type:**	**BY-STEP**
5	**Step size:**	**1**
6	**Rating of undesirability:**	**4**
7	**Reason for undesirability:**	**CHANGES MINOR EQUIPMENT SELECTION/SIZING**

The aspects on the left are pre-defined by SALT. The values on the right are supplied by the domain expert, chosen from a range of values predefined by SALT. Often, the task languages used in method-based tools are somewhat customizable; the tool helps a domain expert specialize the language to *their domain.*

Early expert system development tools, such as OPS5, did not define a task language. The domain expert and knowledge engineer had to come up with suitable notations themselves. The full burden of mapping from those notations to the tool's representations (for example, productions) was left to the knowledge engineer. This approach has the advantage that the tool's scope is not restricted by its task language. But the disadvantages are severe. Since no particular task-level language is defined, the tool cannot provide any assistance in mapping the language to computer representations. Also, the suitability of the chosen external representations is limited by the creativity and experience of the domain expert and knowledge engineer. In practice, this often led to no external notation at all. For example, for a rule-based tool, the domain expert would be asked to articulate knowledge as a set of rules, which the knowledge engineer would rewrite in the tool's formal syntax. The lack of a task-level representation made the resulting expert systems difficult to understand, debug, and maintain.

TAQL is to be able to encode a very broad range of tasks, and so must have a very expressive task language. I chose unrestricted natural language as TAQL's task language. It has the great advantage that the domain expert does not have to learn a new language to describe the task, and is not constrained by an artificial vocabulary of concepts and methods. The drawback is that one cannot fully identify and automate the functions that map from this task language to PSCM — to do so would involve natural-language understanding well beyond the state of the art. Instead, the TAQL programmer must comprehend the natural language, and we (the TAQL developers) attempt to build tools that will make it easier for the programmer to perform the mapping.

By natural language, I mean both prose and the other items that normally appear in documents. For example, a natural-language task description will normally contain some figures and tables, and will use formal notations to some extent.

In choosing natural language as the task language, I am not falling back to the approach of early tools. Those tools did not define any task language at all. Choosing a task language, even one as unrestricted as natural language, lets one attempt to build tools that assist in mapping the task language to computer representations. While that mapping clearly cannot currently be fully automated for natural language, it is possible to identify some aspects of mapping natural language to PSCM, as described in the next section.

In this dissertation, I will often speak of *domain-oriented* task descriptions. These are natural-language descriptions of how a person would perform some task, expressed in the common terminology of the domain. A natural-language task description need not be domain-oriented. For example, a description could present a computation-oriented formulation of a task that bears little resemblance to the strategies and concepts actually used by people in the domain. The distinction is important. Domain-oriented descriptions form the basis of a tool-evaluation methodology that permits unbiased comparison of expert-system development tools. Chapter 3 details this methodology. Briefly, an experimenter gives a subject a domain-oriented task description and asks the subject to implement the task using a specified development tool. The development sessions are logged and the total development time is recorded. Since the descriptions are presented in domain terms, independent of any particular computational formalism, the results of such experiments using different tools should be comparable.

Section 3.1 has more to say about domain-oriented task descriptions, and Appendix C presents a complete domain-oriented description for a small expert-system task.

2.4. Mapping task descriptions to PSCM

Sections 2.1 and 2.2 described PSCM and TAQL, and Section 2.3 described the domain-oriented natural-language descriptions used to describe tasks. Only one of the five central knowledge acquisition tool components listed in Section 1.4 remains to be described: the mapping from task descriptions to the concepts of PSCM and finally to the formal syntax of TAQL.[1]

[1]The material in this section appeared previously in (Yost and Newell, 1989).

Task knowledge is mapped into the TAQL language by a knowledge engineer who comprehends the domain-oriented task description, maps the task concepts to components of PSCM, and composes a set of TAQL statements expressing those PSCM components. For PSCM, the mapping between domain and computational model performs three functions: *identify* a PSCM component; *represent* a data object; and *communicate* some information from one PSCM component to another.

As stated earlier, completely identifying the functions that map from natural language to PSCM is well beyond the state of the art. Nonetheless, the identification of the three informally-specified functions that will be described here is quite useful. It indicates that these are among the activities a knowledge engineer performs when developing a system in PSCM. Therefore, designing programming tools that facilitate these activities should be profitable.

I first describe the three functions in more detail, and then give an example of their application in Section 2.4.1. Let E denote a domain-oriented description of the task knowledge for some task. The first function to be performed is to identify PSCM components in E. All types of PSCM components are identified at this stage, including organizational components such as problem spaces; data-object components that make up the problem-solving state; and problem-solving methods, which determine the behavior of an entire set of PSCM components. The identification proceeds by labeling paragraphs, sentences, or phrases in E with the PSCM components that will encode the knowledge in those parts of E. In essence, it involves segmentation of the text in E.

The labels are assigned based on comprehension of the functional roles of parts of E. For example, a description of how to perform some subtask would be labeled with the name of an operator to perform that subtask, and would be classified as an operator-application component. Components that act together to perform a subtask are grouped into problem spaces. A structure that is the target of some action described in E is identified as a data object (part of a state). The identification of a data object may be further refined by classifying it as an instance of an abstract data type: such an identification is made when E describes manipulations of the identified data object that match the computational operations defined on an abstract data type that is known to the knowledge engineer.

After identification, the next function to be performed in the mapping of E is to *represent* data objects. The identification function yields a conceptualization of the task knowledge in terms of abstract problem spaces. At this stage, most of the procedural structure of the final PSCM solution has been determined. Subtasks have been assigned to operators, operators that directly contribute to performing the same subtask have been grouped into problem spaces, and the relationships among problem spaces have been determined at an abstract level. The interactions among operators within a space are also known at an abstract level. However, the interactions among problem spaces and operators cannot be completely determined until data representations have been selected. Immediately after identification has completed, objects are still in terms of the task domain, except for the occasional appearance of abstract data type terms.

Data representations require raw materials out of which they can be constructed. The choice of raw materials depends to some extent on what is appropriate for the computational model. For example, representations built from machine-level units such as bytes (for instance, records and arrays) have proved appropriate for the computational models underlying conventional programming languages such as C and Pascal. For PSCM, the representations are in terms of attribute/value structures, which have historically proved useful in computational models for expert systems (for example, OPS5 (Forgy, 1981; Brownston, Farrell, Kant, and Martin, 1985)) and are used in Soar.

The representations of the data objects described in E are developed in the same way as the PSCM components were assembled. That is, the attribute/value structures are identified from the structure of their descriptions in E. Thus, if E mentions a cabinet with nine shelves, it might be represented as an object of type *cabinet* with a *shelves* attribute whose value is *9*. These structures can be hierarchical. For example, if E mentions individual cabinet shelves and their heights, the cabinet object may be given a *shelf* attribute, the value of which is an object of type *shelf* having a *height* attribute. When it is clear that a data object corresponds to a familiar abstract data type (for example, a list of items), representation is even simpler, since experienced programmers know how to represent common data types.

Once the representation function has been completed, the re-expression of E in terms of PSCM is almost finished. Most of the interactions among PSCM components are known by the time the identification function completes, and the components need only be restated in terms of the chosen attribute/value representations. However, since the components were identified at an abstract level (before data representations were known), some of these components may now need to be modified or refined.

This fine-tuning of interactions is the province of the *communicate* function. Communication comes in two forms: inter-space communication, and inter-operator communication. Both forms of communication are driven by the need to make available the information operators must have to apply correctly and in the proper order.

The abstract problem space descriptions classify the relations among problem spaces. For example, they may state that the problem space in a subgoal implements a specific operator in the superspace. However, they do not indicate in any detail what information in the current goal needs to be made available in the subgoal, or what information produced by the subgoal needs to be returned to the supergoal when the subgoal exits. The *communicate* function fills in the details of this inter-space communication.

For inter-space communication, data objects are copied from a problem space to a subgoal to make them readily available to the subgoal's problem space. This is particularly important for data objects that are modified by operators in the subspace. Data objects are copied from a subgoal to the superspace either to make a result available to operators in the superspace, or to preserve the value of a data object for a future invocation of the same problem space or one of its subspaces.

For inter-operator communication, there are two distinct situations:

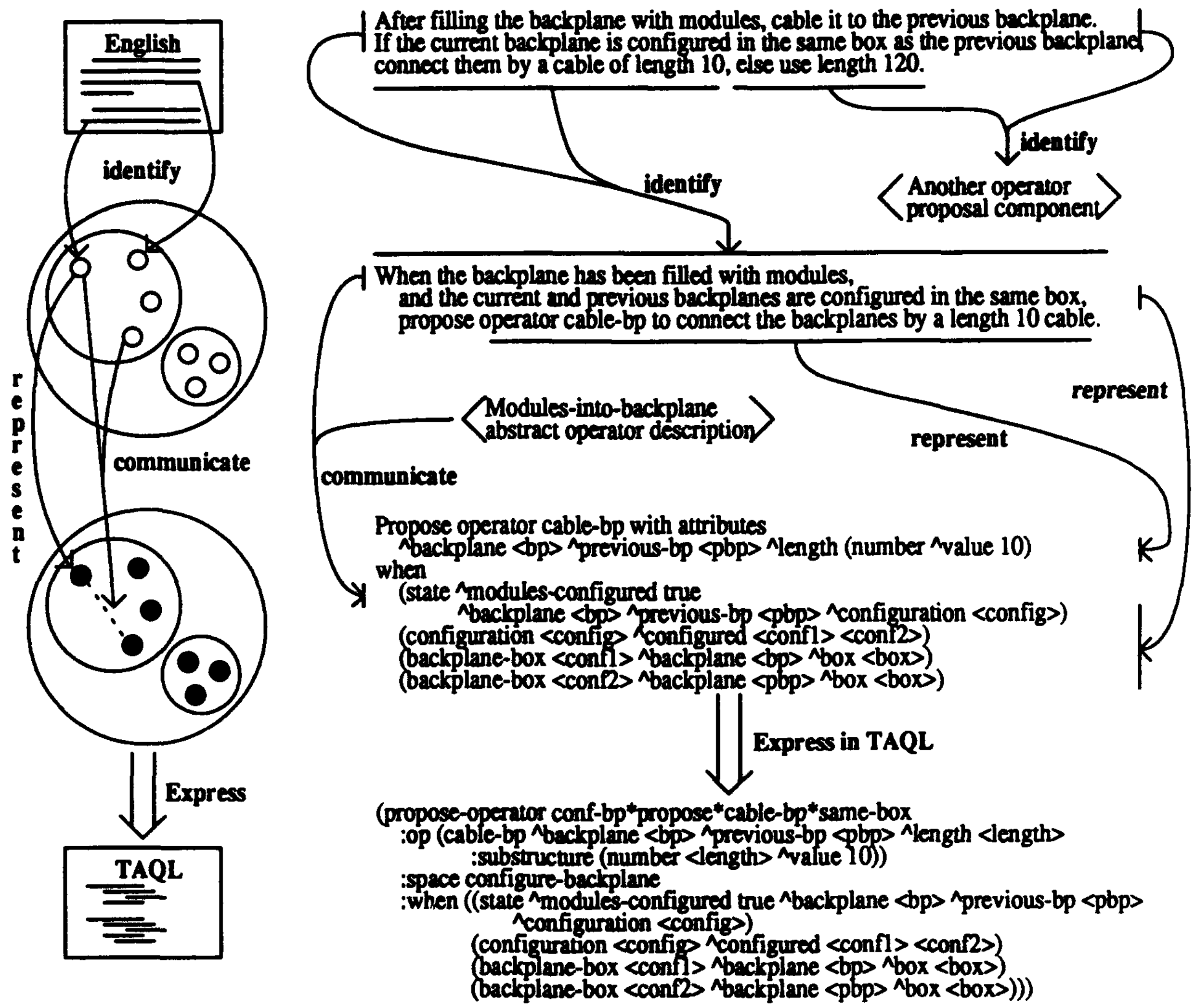
1. When an operator needs data in a form other than the form created by the operator that produces the data.
2. When an operator needs data that was available at some point during prior computation, but that would not otherwise be preserved in the current state.

The first situation can be resolved by either modifying the operators involved so that they represent the data in the same way, or by introducing a new operator or entailment component that translates between the two forms. The second situation can be resolved by modifying operators that had access to the required information in the past so that they make this information part of their result states, thus preserving it for future use.

2.4.1. An example: mapping R1 to TAQL

Figure 2-2 *illustrates how these mapping functions* apply to a small piece of a domain-oriented task description. The domain is computer configuration, as performed by the R1 expert system (McDermott, 1982). R1 was coded in OPS5. Several years ago, the unibus-configuration subtask of R1 was recoded in Soar (Rosenbloom, et al., 1985). R1-Soar is an expert system of about 340 rules. Since its creation, it has served as a testbed for a number of efforts within the Soar project.

I have produced a domain-oriented natural-language description of the unibus-configuration task, and have realized this task in TAQL by applying the mapping functions to that description. The two sentences at the top of Figure 2-2 express when specific instances of an action (backplane cabling) should be performed. This is exactly the kind of information a PSCM operator-proposal component expresses. Thus, the identification function yields two operator-proposal components, one for each of the two cable lengths (only the component for cables of length 10 is shown in detail in the figure). Next, the representation function applies to the conditions in the abstract component, and also to the operator object that is to be proposed. A straightforward mapping from the structure of the abstract component yields the attribute/value representations shown. (Part of the representation of the configuration structure is determined by other parts of the R1 task description that are not shown, and is simply reused here.)



After filling the backplane with modules, cable it to the previous backplane.
If the current backplane is configured in the same box as the previous backplane
connect them by a cable of length 10, else use length 120.

When the backplane has been filled with modules,
and the current and previous backplanes are configured in the same box,
propose operator cable-bp to connect the backplanes by a length 10 cable.

```
Propose operator cable-bp with attributes
  ^backplane <bp> ^previous-bp <pbp> ^length (number ^value 10)
when
  (state ^modules-configured true
         ^backplane <bp> ^previous-bp <pbp> ^configuration <config>)
  (configuration <config> ^configured <conf1> <conf2>)
  (backplane-box <conf1> ^backplane <bp> ^box <box>)
  (backplane-box <conf2> ^backplane <pbp> ^box <box>)
```

```
(propose-operator conf-bp*propose*cable-bp*same-box
   :op (cable-bp ^backplane <bp> ^previous-bp <pbp> ^length <length>
        :substructure (number <length> ^value 10))
   :space configure-backplane
   :when ((state ^modules-configured true ^backplane <bp> ^previous-bp <pbp>
                 ^configuration <config>)
          (configuration <config> ^configured <conf1> <conf2>)
          (backplane-box <conf1> ^backplane <bp> ^box <box>)
          (backplane-box <conf2> ^backplane <pbp> ^box <box>)))
```

Figure 2-2: Mapping part of R1 to TAQL

The condition that determines whether or not the backplane has been filled with modules is easily expressed as a test for the presence of a *^modules-configured* attribute on the state. However, the initial version of the modules-into-backplane operator, which fills the backplane[2], does not generate this information. Thus, the communication function must build a link between the modules-into-backplane and cable-bp operators. It does so by modifying modules-into-backplane to return the required modules-configured attribute, in addition to any other actions the operator already performs.

[2]This operator is described in a part of the R1 description not shown here.

Figure 2-2 shows how the completed PSCM structure maps to TAQL. Because TAQL corresponds so closely to PSCM, the completed PSCM structure and the corresponding TAQL construct are very similar, and mapping from one to the other is straightforward.

Before leaving this example, I say a few words about how R1-TAQL compares to R1-Soar. For the comparisons given here, I use an updated version of R1-Soar that reflects the task-oriented conceptual structure of unibus configuration more closely than the original R1-Soar did.[3]

Both R1-TAQL and R1-Soar use the same seven problem spaces. R1-TAQL has 153 TCs, and R1-Soar has 337 hand-coded Soar productions. The 153 TCs compile into 352 Soar productions. A more useful measure of size is the number of *words* in each implementation, where a word is defined to be the smallest unit that has meaning to the TAQL compiler or to the Soar production compiler. Words include attribute names, variables, and parentheses, among other things. The domain-oriented task description of the part of R1 implemented by R1-Soar has 756 words; R1-TAQL has 5774 words, and R1-Soar has 21752 words. Thus the number of words in R1-TAQL is 26% of the number of words in R1-Soar, a significant reduction.

An important difference between R1-TAQL and R1-Soar is that in R1-TAQL, it is very clear from the code what PSCM role each TC plays. In R1-Soar, the PSCM role of each production is harder to determine. This is because nothing in the Soar productions syntax distinguishes productions based on PSCM-level features. In fact, it is quite possible to write a single Soar production that performs a variety of PSCM roles. The lack of a principled and enforced correspondence between Soar's language (productions) and its computational model (PSCM) makes writing and understanding Soar programs more difficult than necessary, and TAQL attempts to remedy this problem.

As stated earlier, the three mapping functions informally described in this section surely do not completely constitute the mapping from domain-oriented task descriptions to PSCM — such a complete characterization is well beyond the state of the art. Nonetheless, even a partial understanding of the mapping enables designing programming tools that facilitate the mapping.

The rough characterization of the mapping also provides an insight about PSCM. All three functions are guided rather directly by the forms of expression that appear in the domain-oriented task description. The identification function segments the task into problem spaces and operators in a way that mirrors the way the task's actions are described. The representation function chooses data representations that model the way

[3]This was joint work with Amy Unruh.

domain objects are described. The PSCM knowledge roles are mapped directly from small portions of the text, without requiring the prior design of a global problem solving method. This is a direct consequence of PSCM's structure: it makes all decisions locally, purely on the basis of immediately available knowledge. Global behavior arises from these local decisions, and so does not need to be pre-designed or pre-specified.

2.5. Departures from the standard method-based approach

The ultimate knowledge acquisition tool would be able to acquire a task in any domain directly from a domain expert. There are two absolutes here: that the tool be able to acquire any task, and that it acquire it directly from a domain expert, who might not be a programmer.

Achieving both absolutes in a single tool is well beyond the state of the art. The limitation arises as follows. If a non-programmer is to use the tool, then the mapping from domain knowledge to computational terms must be done entirely by the tool itself, for it is this mapping that constitutes computer programming. But if there are no constraints on the form in which domain knowledge is expressed, the tool cannot perform the mapping. That would imply human-level language comprehension abilities. So the domain language must be constrained, and this in turn constrains the range of tasks to which the tool can be applied.

Traditional method-based knowledge acquisition tools adopt the second absolute — that the tool be usable by a non-programming domain expert. Each tool is thus limited to a narrow range of tasks. The research strategy involves gradually widening the achievable range.

My research has the same ultimate goal, but adopts the other absolute as fixed: that the tool be able to acquire a full range of tasks. The research strategy involves automating and facilitating more and more of the mapping so that less and less programming skill is needed. To that end, I have selected a general-purpose problem-solving method (PSCM) and an unconstrained domain-description language (natural language). I have identified some important mapping functions and attempted to build tools to automate or facilitate them.

My thesis is that the concepts and design strategies used in traditional method-based tool research can be applied very profitably in this inverted approach. As shown in Chapter 5, the resulting tool (TAQL) is very broadly applicable and is more effective than traditional method-based tools, even for tasks that are very well-suited to those tools.

2.5.1. Embedding methods and method-based tools in Soar

Each problem space in a PSCM system can be viewed as behaving according to a method that is more specialized than PSCM itself. Each addition of knowledge to a problem space further constrains the behaviors it can produces. In particular, by adding the right knowledge, one can construct problem spaces that behave according to the methods that have been used in traditional method-based tools. Thus, one way to extend TAQL so that less programming skill is required would be to embed methods from traditional method-based tools in Soar. For appropriate tasks, knowledge engineers could then simply insert domain knowledge into these pre-defined methods.

Johnson, Smith, and Chandrasekaran describe such an effort (Johnson et al., 1989; Johnston, et al., 1990; Johnson, 1991). They show how to embed the generic-tasks approach to method-based knowledge acquisition (Chandrasekaran, 1986) in Soar.[4] This section briefly presents their work, its motivations, and its results.

A generic task (*GT*) couples a type of task with an appropriate problem-solving method and the kinds of knowledge needed to use the method (the method's knowledge roles, in the terminology of this dissertation). Chandrasekaran (Chandrasekaran, 1986) describes a variety of GTs for task types including hierarchical classification, hypothesis matching, abductive assembly, and qualitative simulation.

GTs are a classic example of method-based knowledge acquisition, and share the advantages of that class. GTs also share the disadvantages characteristic of the method-based approach. Having a narrow, predetermined problem-solving method is inconvenient for tasks that would more naturally use a slight variant of the method. This makes GTs brittle and inflexible. It is also difficult to build systems for tasks that have many subtasks, and hence require multiple methods. The modules embodying each method do not share a common representational base, and the types of communication allowed between methods are fixed at the time the module is designed. Furthermore, new methods cannot be constructed easily by combining parts of existing methods.

Johnson and colleagues reasoned that embedding GTs in an appropriate general architecture would overcome these problems while retaining the advantages of existing GTs. They chose Soar as the general architecture, and found that GTs are easily embedded in Soar. PSCM's initial state and goal test knowledge represent the task type. Operators and search control (selection) knowledge embody the problem-solving method. The kinds of knowledge needed to use the method are represented on the PSCM state and in the operators that manipulate those parts of the state.

[4]This work was done using Soar directly, and has not yet been incorporated into TAQL. However, the work was done at the PSCM level, and I foresee no difficulties in doing so.

Embedding GTs in PSCM overcomes the problems listed above. In the PSCM approach, a GT's method only specifies the structure central to the GT's definition. For example, in hierarchical classification, the order in which hypotheses are refined is not central to the definition. Thus the PSCM GT does not specify selection knowledge for the operators that refine hypotheses. Consequently, the task developer is free to implement whatever control strategy is desired. If the developer does not supply a strategy, the system will fall back on PSCM's default weak methods. This makes the PSCM GT more flexible and robust than the non-PSCM GT.

Since PSCM provides a common representational base for GTs, it is much easier to build systems that use multiple communicating GTs. GTs can easily and flexibly share information, and new methods are readily formed by combining operators from existing methods in novel ways.

PSCM GTs have been implemented for hierarchical classification, abductive assembly, and hypothesis matching. Also implemented is Red-Soar (Johnson et al., 1991), an antibody identification expert system that mixes subtasks from several PSCM GTs in a single system.

(Johnson et al., 1989) describes ER-Soar, the Soar implementation of the hierarchical classification GT. I close this section by quoting from the discussion section of that paper:

> ER-Soar combines the advantages of the GT approach with the advantages of the Soar architecture. Knowledge acquisition, ease of use, and explanation are all facilitated in ER-Soar because subgoals of the problem solving method and the kinds of knowledge needed to use the method are explicitly represented in the implementation. The subgoals of the method are directly represented as problem space operators. The kinds of knowledge needed to use the method are either encoded in productions or computed in a subgoal. The same advantages apply to the supplied methods for achieving subgoals. Finally, the implementation mirrors the GT specification quite closely making ER-Soar easy to understand and use.
>
> ER-Soar is able to overcome many of the problems suffered by previous GT systems. [Soar's] Automatic subgoaling allows unanticipated situations to be detected and handled. If no specific method for handling the situation is available, an appropriate weak method can be used. Whenever a goal needs to be achieved it is done by first suggesting problem spaces and then selecting one to use. This allows new methods in the form of problem spaces to be easily added to existing problem solvers. Once again, if no specific technique exists to determine which method to use, Soar will try to pick one using a weak method. [Soar's] Automatic subgoal termination provides an integration functionality not available in previous GT architectures. In general, the integration capabilities of ER-Soar are greatly enhanced. Because of [Soar's] preferences and the additive nature of productions, new knowledge can be added to integrate ER-Soar with other methods. None of ER-Soar's control knowledge needs to be changed when the new knowledge is added.

2.6. Where we stand

This chapter described how the design strategies traditionally used to develop method-based tools apply in the context of Soar. Each section in this chapter has described how one of the central tool components listed in Section 1.4 is instantiated for Soar. Section 2.1 described a general-purpose problem-solving method, PSCM — the method that underlies Soar. Section 2.2 presented TAQL, a computer language that directly expresses PSCM computations. Section 2.3 described and justified the selection of domain-oriented natural language as the task-description language. Finally, Section 2.4 discussed the mapping from domain-oriented task descriptions to the concepts of PSCM and ultimately to TAQL.

The remainder of the dissertation evaluates TAQL's adequacy. I show that TAQL compares very favorably to existing tools. This supports my thesis that the strategies used to design method-based tools are beneficial even when applied to a weak problem-solving method.

Chapter 3
Evaluation Methods

Since its inception in 1986, the Banff Knowledge Acquisition for Knowledge-Based Systems Workshop has emerged as the field's premier conference. The 1990 proceedings included 36 papers by authors from around the world, 24 of which described operational tools or techniques. Of these tools and techniques, fourteen can be used to construct a running system from scratch. Only three of these fourteen papers provided any quantitative performance analysis. None of the three evaluated the tool or technique on more than three test cases. Two of the three papers were missing critical quantities — for example, they report how long it took to build an application with the tool, but do not report the application's size, or vice versa. The third paper gave complete data, but only for one case study.

This same scarcity of data and lack of rigorous evaluation pervades the expert-systems and knowledge-acquisition fields. I single out the Banff workshop only to show that the problem is severe even in the highest-quality research forums. The lack of such data makes it impossible to do credible comparative evaluations of different knowledge acquisition tools and techniques.

When a field is small, the lack of data may not be so problematic. Researchers are more likely to know each other and to have personal experience with a reasonable fraction of the field's approaches. The knowledge acquisition field has grown to a point where this is no longer true. Tools and techniques abound, and it is impossible for a given researcher to have extensive first-hand experience with a significant fraction of them. Unless rigorous evaluation becomes commonplace, researchers will be increasingly unable to select profitable avenues for future investigation.

Brian Gaines, organizer of the Banff workshop, listed poor evaluation as one of the field's impediments at the 1987 workshop. At the 1990 workshop, he says (Gaines, 1990, p. 8-2):

> In 1987 I went on to say, "These are not unreasonable problems at this stage of development of knowledge based systems." In 1990 I would like to propose to this meeting that these problems are becoming highly unreasonable. We should accept them as the major research challenges for the 1990s, and develop an action plan to overcome them. They are challenges not only for us as individual researchers, but also

for the knowledge acquisition community as a whole. Indeed many of the problems are essentially ones for the community not the individual — we can all contribute but no one of us alone can solve them.

This chapter describes the evaluation methodology that has been developed for TAQL. The methodology is not flawless, and does not yield all the information one can imagine wanting. Nonetheless, it is simple to apply and yields a great deal of useful data.

My philosophy in designing an evaluation methodology has been to adopt the 80-20 rule: that 80% of the benefits are often obtained from only 20% of the effort (and, conversely, that the remaining 20% of benefit requires 80% of the work). This rule applies both to developing evaluation techniques and to applying them. In the first case, one should not spend so much time trying to develop perfect evaluation techniques that no evaluation is ever actually performed. In the second case, evaluation techniques should yield substantial useful data and yet be simple enough to apply that practitioners will not resist using them.

It is tempting not to follow the 80-20 rule, but the result of following that path is that no evaluation is performed. For instance, Dhaliwal and Benbasat (Dhaliwal & Benbasat, 1990) present a framework for comparative evaluation of knowledge acquisition tools and techniques. The paper is insightful and comprehensive. It lists six dependent variables, four independent variables, and eighteen moderator variables. Developing an evaluation methodology that accounts *for all of these factors* would be extremely difficult. Even if it could be done, it would very likely be so difficult to measure all of the factors that only very diligent, well-staffed research groups would use the methodology. Dhaliwal and Benbasat do not present an example of their framework's use, and although the paper is recent it does not seem likely that their framework will see significant use, if it is ever used at all.

3.1. Experiment design

I evaluated TAQL's effectiveness in a series of software-development experiments. In each experiment trial, a subject was given a domain-oriented description of some task, and was asked to develop an initial prototype implementation using TAQL. It was up to the subject to decide when the prototype was satisfactorily completed. The code did not have to be particularly clean or efficient or have a good interface, but it did have to perform correctly on at least the test cases presented in the task description.

Subjects kept logs of their development activities. Each log included brief time-stamped entries for any events the subject felt were significant, including at least

- Beginning and ending times for task understanding (reading the description), design, coding, and testing/debugging. (A development session usually contains many episodes of each of these activities.)

- Bug information: the symptoms that let the subject know there was an error, the nature of the error, and when and how it was fixed.

Also recorded were the size of the subject's implementation in terms of the number of TCs, the number of Soar productions those TCs compiled into, the number of symbols in the source code (the TCs), and the number of symbols in the compiled Soar productions. Dividing the total development time by the system size gives a measure of the subject's average encoding rate. The magnitude of this rate gives a rough measure of TAQL's overall effectiveness.

The bug data gives more detailed information on TAQL's shortcomings. From it, one can form error classifications and determine which kinds of errors are accounting for the most development time. Of all the data that was collected, the bug data provided the strongest guidance in improving TAQL. Section 4.4 discusses this further.

The Soar architecture supports learning. For the TAQL experiments, subjects were told to keep learning turned off unless they felt that there was some very compelling reason to turn it on in a particular task. The reason is that the nature of the problem space representations used in a Soar or TAQL system strongly influences the quality and correctness of the new productions it learns. The kinds of representations that result in correct learning are not yet well-understood. Soar users often report that building a system that learns correctly is *much* harder than building a system that performs correctly without learning. Learning was turned off during the experiments because I did not want this poorly-understood aspect of Soar to confound the results.[5]

It is important to understand the nature of the task descriptions the subjects used. They are natural, *domain-oriented* presentations of what the task is and how a person could perform it.[6] They include all the detail a person would need to be able to understand and perform the task, and no more. They are not implementation-oriented specifications for a computer program. For example, consider these sentences taken from a description of how to estimate the construction time of a high-rise building:

> If the building does not have a basement, then the construction sequence is to build the footings, then the lobby structure, then the tower structure, all using the typical floor formwork crew size. Fewer than 75% of the typical crew are likely to be needed for the lobby structure, but the full crew is assumed to be available if needed.

While the first sentence describes processes involved in performing the task and how

[5]Subjects did turn learning on in two tasks. See the description of chunking errors in Section E.1 for details.

[6]Some of the tasks in the experiments are puzzle tasks. Puzzles by definition define only the problem and not how to solve it.

they are sequenced, it does not do so in implementation-level terms. The knowledge is expressed solely in the terminology of the domain. This has an important implication for the experiments: since the descriptions are not tied to any particular computational formalism, running the same experiments with different expert-system development tools would permit unbiased comparison of those tools. Section 2.3 contains additional information on domain-oriented task descriptions, and Appendix C presents a complete domain-oriented description for one of the small expert-system tasks used in the TAQL experiments.

3.2. Discussion

My thesis work is primarily concerned with evaluating tools on two dimensions: scope and effectiveness. The experiment design described in the last section permits evaluating both of these.

To evaluate scope, one must choose experiment tasks that cover many different task types and sizes. A tool's scope is characterized by the set of tasks it is able to implement. Section 4.1 lists the tasks that were used in the TAQL experiments, and Section 5.1 evaluates TAQL's scope.

The effectiveness measures are based on the total time it takes to develop tasks in the experiments. Times are collected for four different aspects of development: task understanding, design, coding, and testing/debugging. This permits separately analyzing a tool's effectiveness on each of these dimensions.

Total development time is a useful measure for comparing the effectiveness of different tools for the same task. Another measure is more useful for analyzing the effectiveness of a single tool for different kinds of tasks: the *encoding rate*. The encoding rate is a measure of time per unit encoded. Possible measures for TAQL are time per TC, time per Soar production, time per symbol of source code, and time per symbol of compiled code (productions). Section 5.2.2 analyzes the encoding rates for the TAQL experiments and the relative quality of the four encoding rate measures (Section 4.3.1 also analyzes the relative quality of different measures of system size). This document typically uses the time-per-Soar-production measure. It proved to be at least as good as the other three measures, and number of productions is a common measure of system size in the Soar community and elsewhere. One cannot use encoding rates to compare tools that use different representational bases. This is true even for systems that use superficially similar representations, such as OPS5 and Soar — although both use productions, the expressive power of a single production is likely to be different for the two languages.

The domain-oriented task descriptions are the key to enabling reliable comparisons among tools. Since they are written in the terminology and methods of the domain rather than in terms of any particular computational model, they are not biased unfairly in favor of any particular tool. Thus, one can meaningfully compare the development times for the same task using different tools. I would have liked to have used more descriptions written by other people in the TAQL experiments. Unfortunately, I could find only one expert-system task description written by someone else, and had to write the rest myself. Of the few detailed descriptions I did find in the literature, all were either missing parts of the domain knowledge or were expressed in computational terms. I urge expert system developers and researchers to write such descriptions and make them available. Simply providing data on the development effort is a step in the right description, but it is not enough. We as researchers need a large and varied set of tasks with which to evaluate our tools.

The experimental design presented here does not control for a variety of factors that could influence the results. The most important of these factors are:

- *Programmer skill and background.* The programmer's experience with the tool and with programming in general can have a great impact on development time, as can prior familiarity with the task domain.
- *Appropriateness of the task for the tool.* Although the domain-oriented descriptions are not deliberately biased in favor of any particular tool, a tool will nonetheless be better-suited to some tasks than others.
- *Programmers may learn during the experiments.* This is particularly true if the programmer is not already an expert user of the tool.
- *Subjectivity in the stopping criteria.* Programmers are left to decide for themselves when the prototype is good enough to be considered finished. The prototype must correctly run a set of test cases included in the task description. This still leaves considerable leeway in deciding when the program is sufficiently clear, general, and efficient, however.

One might worry that these sources of variability will result in relatively random task-development data. That fear is not born out by the data collected for TAQL. In spite of the potential flaws, the data shows strong regularities, as described in Sections 4.3.1 and 5.2. Furthermore, all of these effects can be expected to cancel out when data is available for many different tasks and programmers. Hopefully, other researchers will use the methodology presented here, thus extending and improving the pool of data. A larger pool of data will also make it easier to detect anomalous entries that deserves further examination.

One can also alleviate the effects of variability by reporting departures from the norm in the experiments. For example, experimenters should report a programmer's level of experience with the tool and any prior familiarity with the domain. Doing so helps the reader more accurately interpret the data. The more information experimenters report about moderating influences in their data, the more likely it is that other researchers will be able to meaningfully compare their results to it.

In summary, I believe that the evaluation procedures described in this chapter provide a great deal of useful information with little effort. Creating the experiment logs requires only jotting down a small amount of information over the course of development. Furthermore, an experimenter does not have to be present. The subject is free to work as he or she normally would. Because the logs are short, they are easy to analyze. Therefore running an experiment requires little effort for both the subjects and the experimenter.

Chapter 4
The Experiments

There were three rounds of TAQL task-development experiments, using three different versions of TAQL. Round 1 was in January and February 1990, and used TAQL 3.1.0 running under Soar 5.1.1. Round 2 was held September through December, 1990, and used TAQL 3.1.3 running under Soar 5.2.0. Round 3 was in May and June 1991, and used TAQL 3.1.4 running under Soar 5.2.0. Rounds 1 and 2 used pre-release versions of TAQL. Round 3 was actually two mini-rounds, using two different versions of TAQL 3.1.4. TAQL's evolution between rounds was almost exclusively based on the analysis of the subjects' experiment logs from the prior round (some changes were made just because they seemed like good ideas).

Over the course of the experiments, three subjects developed 31 systems drawn from a pool of 19 tasks. No subject worked on the same task more than once. All three subjects participated in the first two rounds, but only one subject participated in Round 3 due to the time required to develop the large tasks in that round. The average task size increased substantially in each round. Round 1 used mostly small puzzle tasks, Round 2 used mostly small expert-system tasks, and Round 3 used mostly large expert-system tasks.

The remainder of this chapter describes the experiment tasks and subjects, and presents the data gathered. In addition, Section 4.4 describes the versions of TAQL used in the experiments and how and why TAQL evolved between rounds.

4.1. The tasks

The experiments covered 19 different tasks. Ten of these were puzzle tasks, and the other nine were expert-system tasks. The puzzle task descriptions just define the problems, and do not give hints as to how to solve them. The expert-system task descriptions define the problem to be solved and describe in domain terms the knowledge and strategies a domain expert could use to perform the task. Seven of the nine expert system task descriptions also include at least one sample test case with its solution. For the other two, it is easy for the developers to create their own test cases. I describe each task briefly below.

Table 4-1 defines abbreviations I will use to refer to the experiment tasks, and also lists the number of words in each task description. The abbreviations also appear in the descriptions below.

Puzzle tasks:

- *Beautiful words [BeaW]* (Newell, McDermott and Forgy, 1977, p. 49): Form as many English words as possible from the letters in the word *beautiful.*
- *Traveling salesperson [Trav]* (Newell, McDermott and Forgy, 1977, p. 96): A salesperson must visit each of a set of cities exactly once. Find a visiting order that minimizes the total distance traveled.
- *Missionaries and cannibals [MaC]* (Newell, McDermott and Forgy, 1977, p. 63): Three missionaries and three cannibals must cross a river in a two-person boat. Find a sequence of boat crossings that gets everyone across the river without the cannibals ever outnumbering the missionaries on either side of the river.
- *Republicans and democrats [RaD]* (Smullyan, 1982, p. 6): Given some statements about the relative proportions of republicans and democrats in a group of people when some people change affiliations, deduce the total number of people in the group.
- *Monkey and bananas [MaB]* (Newell, McDermott and Forgy, 1977, p. 70): A monkey is in a room containing a bunch of bananas and a box. The bananas are suspended from the ceiling. The monkey can move around the room, push the box, climb on and off the box, and reach for the bananas, but is too short to reach the bananas when standing on the floor. Find a way for the monkey to get the bananas.
- *Sequence extrapolation [Seq]* (Newell, McDermott and Forgy, 1977, p. 82): Given a sequence of integers, predict the next number(s) in the sequence. The given sequence can be a constant sequence, an arithmetic progression, a geometric progression, or some combination of these.
- *Towers of Hanoi [Hanoi]* (Newell, McDermott and Forgy, 1977, p. 69): Three disks are stacked on one of three pegs, with each disk being smaller than the one it rests on. Moving one disk at a time, transfer all three disks to the third peg, without ever stacking a disk on a smaller disk.
- *Picnic problem [Pic]* (Newell, McDermott and Forgy, 1977, p. 27): Given assorted facts about how three boys spent their money on picnic supplies, deduce how much one of the boys spent on soda pop.
- *Kings and wizards [KaW]* (Newell, McDermott and Forgy, 1977, p. 35): A king paints a black or white spot on the forehead of each of three wizards. The wizards are told that at least one of the spots is white. The first wizard to correctly guess the color of his own spot will be rewarded, and the other two will be put to death. After a few seconds, one of the wizards guesses correctly. How did he do it?
- *Island of questioners [IofQ]* (Smullyan, 1982, p. 63, questions 1 through 5): An island is inhabited by people who can only speak in questions. Some of

them only ask questions whose answer is *yes*, and the others only ask questions whose answer is *no*. For each of five scenarios involving a question, determine what can be deduced about the class of the people in the scenario.

Expert system tasks:

- *Material handling equipment selection [Mater]* (Yost, 1992b): Given information about the type of material to be moved and other characteristics of the move, suggest appropriate types of material handling equipment (conveyor belt, fork lift, and so on).
- *Bird classification [Birds]* (Yost & Altmann, 1992): A person is observing (or has in mind) a bird that they want to identify. The system must ask the person some questions about the bird's characteristics, and either tell the person the name of the bird or tell them that no bird that it knows of matches their description. The number of questions asked need not be minimal, but unnecessary questions should be avoided to a reasonable extent. (Merritt, 1989) describes a version of this task.
- *Job-shop scheduling [Sched]* (Yost & Altmann, 1992): A job shop accepts objects from customers and processes them so that they have some customer-specified characteristics. The processing must be completed within some customer-specified time limit. The shop has several machines for processing the objects, for example polishers and drills. Schedule a set of jobs on the machines subject to a variety of constraints. (Minton, 1988) describes a version of this task.
- *Shipment scheduling assistant [Truck]* (Yost & Altmann, 1992): A trucking company ships materials among cities in the Midwest. A trip itinerary specifies a travel route and the pickups and deliveries that must be made at cities along the route. Assign trucks and drivers to given itineraries subject to a variety of constraints. (Filman, 1988a) describes a version of this task. Appendix C presents the complete domain-oriented description for this task.
- *Computer system sizing [Sizer]* (Whitney et al., 1990): Given sketchy information about a customer's organization (for example, industry and number of employees of various types), estimate the amounts of various computing resources they will need. Estimates are based on extrapolation from a similar previously-encountered case. (Offutt, 1988) describes a similar formulation of the sizing problem and a knowledge acquisition tool for sizing systems.
- *Oil pipeline scheduling [Pipe]* (Yost, 1992c): Schedule oil pipeline tenders to meet demands at pipeline terminals. Different oil products are pumped in a fixed, predetermined sequence. Given that sequence, determine appropriate product amounts and pumping rates, subject to a variety of tankage and flow rate constraints.
- *EMP hardness evaluation [EMP]* (Yost, 1992d): Evaluate an electro-mechanical design with respect to EMP (electromagnetic pulse) hardness. If the design is not EMP-hard, suggest appropriate design modifications. Generate a report on the design and its evaluation in a standard form.

(Klinker, 1988) discusses the DPR WRINGER, an expert system that performs a version of this task. Klinker also describes KNACK, a knowledge acquisition tool for such evaluation/reporting systems.

- *Construction time prediction [Predict]* (Yost, 1992e): Predict the construction time of a multi-story building in Australia, given sketchy design information. The design information is scant enough that it will be available early on, when rough but somewhat reliable construction time estimates are needed for planning and contracting. The description includes guidelines for estimating construction sequences, required and available workforce sizes, and weather delays, among other things. (Swartwout, 1986) describes a system that performs a version of this task.
- *Elevator configuration [VT]* (Yost, 1992f): Given elevator performance requirements (for example, speed and capacity) and relevant building information, find a suitable elevator-system configuration (one that satisfies the functional requirements and meets the safety code). The task description includes extensive information on elevator components and their interrelationships. It also describes the functional and safety constraints an elevator system must satisfy, and ways to modify designs that do not meet these constraints. (Marcus, Stout & McDermott, 1988) describes VT, a system that performs a version of this task.

4.2. The subjects

Three subjects participated in the TAQL task-development experiments. All three were computer-science doctoral students at Carnegie Mellon University, and all were members of the Soar project.

- S1 has a bachelor's degree in math and computer science. The experiments occurred during his first and second years of graduate school. Prior to the beginning of the first round of the experiments, he had written one small Soar program and translated it into TAQL. Throughout, he was a member of the rapid task acquisition research group (of which TAQL is a part), but he was not a TAQL developer.
- S2 has a bachelor's degree in computer science. The experiments occurred during his third and fourth years of graduate school. He became a TAQL co-developer four months before the experiments began. He had written several Soar programs and a few small TAQL programs before the experiments.
- S3 has a bachelor's degree in math and computer science. The experiments occurred during his fifth and sixth years of graduate school. He is the primary TAQL developer. Prior to the experiments, he had written several sizable Soar programs and several TAQL programs, including one mid-sized expert system.

Task	Name abbreviation	Description size (words)
Beautiful words	BeaW	56
Traveling salesperson	Trav	68
Missionaries and cannibals	MaC	80
Republicans and democrats	RaD	83
Monkey and bananas	MaB	93
Sequence extrapolation	Seq	100
Towers of Hanoi	Hanoi	105
Picnic problem	Pic	109
Kings and wizards	KaW	170
Island of questioners	IofQ	395
Material handling equipment selection	Mater	550
Bird classification	Birds	600
Job-shop scheduling	Sched	750
Shipment scheduling assistant	Truck	1900
Computer system sizing	Sizer	2600
Oil pipeline scheduling	Pipe	4300
EMP hardness evaluation	EMP	6800
Construction time prediction	Predict	11700
Elevator configuration	VT	16281

Table 4-1: TAQL experiment tasks

4.3. The results

Tables 4-2 through 4-4 summarize the data collected in the three experiment rounds, from the logs kept by the subjects.[7] In addition to the data analysis presented in this section, Section 5.2 analyzes the data with respect to TAQL's effectiveness.

The columns in each table are divided into three sections separated by heavy lines, as follows:

[7]Appendix E presents additional experiment data.

1. The first set of columns give a number identifying the table row, the subject who developed the task, and the abbreviated task name (see Table 4-1).
2. The second set of columns give the times the subject spent understanding the task description and designing, coding and testing/debugging their program. These times add up to the total experiment time for each task. The time spent reading and understanding the task description is referred to as *builder task acquisition* in the tables. All times are in minutes. The number in parentheses after each time gives the percent of the total development time spent doing that activity.
3. The third set of columns give summary information:
   - Two measures of system size: the number of TCs (TAQL constructs) in the final system and the number of SPs (Soar productions)[8] compiled from those TCs.
   - The average number of SPs compiled from each TC.
   - Three measures of the subject's encoding rate: the average number of minutes per TC and SP, and the average number of seconds per TC symbol. All of these are computed over the total task development time.

     TC size can vary substantially depending on coding style. For example, a TAQL programmer can often choose to encode several independent pieces of knowledge in a single TC, or to encode each piece of knowledge in a separate TC. The seconds-per-TC-symbol measure gives a measure of encoding rate independent of variations in TC size or the number of productions compiled from a TC. On the other hand, minutes per TC and SP give better measures of the time required to encode basic conceptual units in the language.
   - Two measures of bug frequency: the average number of bugs per hour, and the average number of bugs per TC.
   - A measure of the knowledge density of the task description: the number of source code (TC) symbols in the final system divided by the number of words in the task description.
   - A classification of the number of words in the task description into orders of magnitude. Table 4-1 gives the precise task description sizes.

Table rows are grouped by subject. Within subject, the rows appear in the order the tasks were developed. The tables include average lines for each subject and for each round.

[8]I call these *SPs* or *Soar productions* rather than simply *productions* to emphasize that the productions used in different production system languages may vary widely in expressiveness. Therefore, the number of productions required to implement a system may be very different for other production languages.

Subjects sometimes chose to use representations in which some groups of TCs were nearly identical, differing only in the domain knowledge that instantiated a prototypical TC form. Rather than type all of these TCs in by hand, the subjects typically chose to write simple programs (usually in Lisp) that generated the TCs from a TC template and a data file of domain knowledge. The data in the tables includes the time spent developing and using these helper programs, but the TC, SP, and TC symbol counts include only one instance per template. So, for example, if a system had 200 hand-coded TCs and 100 additional TCs generated from a single template, the TC count for that task would be given as 201, not 300. The notes appearing under each table give the number of tool-generated TCs and SPs for these tasks.

Program-generated TCs are excluded to give a more accurate picture of TAQL's effectiveness. When TC generators are used, the only TAQL-related effort the programmer expends is writing the TC templates that the generator will use. Counting all of the generated TCs would artificially deflate the programmer's encoding rate.

	Who	Task Name	Builder Task Acq	Design	Coding	Test/ Debug	Total (Min)	TCs	SPs	SP/ TC	Min/ TC	Min/ SP	Sec/ TC Sym	Bugs/ Hour	Bugs/ TC	TC Sym/ Desc Wd	Desc Size (Words)
1		MaC	1 (0)	15 (4)	45 (12)	330 (84)	391	18	32	1.8	21.7	12.2	24.7	0.5	0.17	11.9	
2	S1	KaW	2 (1)	40 (11)	80 (22)	243 (67)	365	27	42	1.6	13.5	8.7	22.8	1.3	0.30	5.6	
3		Hanoi	1 (1)	12 (8)	74 (50)	60 (41)	147	17	24	1.4	8.6	6.1	10.0	1.6	0.24	8.4	
4		Seq	10 (2)	86 (13)	361 (55)	196 (30)	653	78	111	1.4	8.4	5.9	14.1	1.5	0.21	27.8	
Averages			**(1)**	**(9)**	**(35)**	**(55)**	**389**	**35**	**52**	**1.5**	**13.1**	**8.2**	**17.9**	**1.2**	**0.23**	**13.4**	
5		KaW	30 (16)	90 (49)	60 (32)	5 (3)	185	20	39	2.0	9.2	4.7	15.7			4.2	
6		BeaW	150 (30)	155 (31)	165 (33)	30 (6)	500	47	93	2.0	10.6	5.4	15.4	0.2	0.04	34.8	~ 100
7	S2	MaB	1 (2)	25 (49)	20 (39)	5 (10)	51	15	23	1.5	3.4	2.2	8.8			3.7	
8		Trav	1 (1)	35 (33)	55 (52)	15 (14)	106	12	43	3.6	8.8	2.5	8.2	0.6	0.08	11.4	
9		RaD	5 (6)	40 (44)	34 (38)	11 (12)	90	20	45	2.2	4.5	2.0	6.5	2.0	0.15	10.0	
10		IofQ	30 (8)	170 (48)	80 (23)	75 (21)	355	56	73	1.3	6.3	4.9	13.2	2.9	0.30	4.1	
Averages			**(11)**	**(42)**	**(36)**	**(11)**	**215**	**28**	**53**	**2.1**	**7.2**	**3.6**	**11.3**	**1.4**	**0.14**	**11.4**	
11		MaB					211	29	46	1.6	7.3	4.6	12.8			10.6	
12	S3	Truck	35 (7)	120 (26)	195 (41)	120 (26)	470	94	133	1.4	5.0	3.5	7.2			2.0	
13		Birds	20 (9)	10 (5)	128 (60)	57 (27)	215	31	74	2.4	6.9	2.9	10.5	1.7	0.19	2.1	~ 1000
14		Sched	60 (13)	106 (24)	87 (19)	194 (43)	447	59	125	2.1	7.6	3.6	7.6	1.5	0.19	4.7	
Averages			**(10)**	**(18)**	**(40)**	**(32)**	**336**	**53**	**95**	**1.9**	**6.7**	**3.7**	**9.5**	**1.6**	**0.19**	**4.9**	
Overall avg.			**(7)**	**(26)**	**(37)**	**(29)**	**299**	**37**	**65**	**1.9**	**8.7**	**4.9**	**12.7**	**1.4**	**0.19**	**10.1**	
			Times in minutes. '(-)' is % of Total.														

Notes, by row:

1	Details of bugs are unclear from the log	11	Log data insufficient to determine time breakdown or bug stats
5,7,12	Log data insufficient to determine bug stats	13	Excludes tool-generated TCs (153 TCs/153 SPs)

Table 4-2: Task development data: Round 1

	Who	Task Name	Builder Task Acq	Design	Coding	Test/ Debug	Total (Min)	TCs	SPs	SP/ TC	Min/ TC	Min/ SP	Sec/ TC Sym	Bugs/ Hour	Bugs/ TC	TC Sym/ Desc Wd	Desc Size (Words)
1	S1	Sched	5 (2)	37 (14)	99 (37)	126 (47)	267	38	89	2.3	7.0	3.0	8.4	0.9	0.11	2.5	
2		Truck	10 (2)	115 (23)	235 (48)	133 (27)	493	66	171	2.6	7.5	2.9	10.2	1.6	0.20	1.5	
3		Mater	15 (16)	15 (16)	50 (53)	15 (16)	95	20	59	3.0	4.7	1.6	5.8	1.9	0.15	1.8	
4		Birds	5 (5)	35 (32)	50 (46)	18 (17)	108	23	32	1.4	4.7	3.4	8.3	1.7	0.13	1.3	
Averages			**(6)**	**(21)**	**(46)**	**(27)**	**241**	**37**	**88**	**2.3**	**6.0**	**2.7**	**8.2**	**1.5**	**0.15**	**1.8**	
5	S2	Truck	25 (5)	175 (35)	145 (29)	150 (30)	495	44	91	2.1	11.2	5.4	11.4	2.7	0.50	1.4	~ 1000
6		Birds	70 (11)	200 (33)	240 (39)	105 (17)	615	66	87	1.3	9.3	7.1	11.3	1.8	0.27	5.5	
7		Sched	45 (8)	150 (26)	165 (28)	220 (38)	580	38	89	2.3	15.3	6.5	18.2	0.8	0.21	2.6	
8		Mater	15 (7)	40 (19)	85 (40)	70 (33)	210	49	114	2.3	4.3	1.8	3.0	4.6	0.33	7.6	
Averages			**(8)**	**(28)**	**(34)**	**(30)**	**475**	**49**	**95**	**2.0**	**10.0**	**5.2**	**11.0**	**2.5**	**0.33**	**4.2**	
9	S3	Mater	12 (10)	18 (16)	70 (61)	15 (13)	115	15	60	4.0	7.7	1.9	5.8	1.0	0.13	2.2	
10		Pipe	110 (11)	208 (21)	345 (34)	343 (34)	1006	116	375	3.2	8.7	2.7	8.7	1.1	0.16	1.6	~ 5000
11		Sizer	65 (16)	130 (32)	135 (33)	73 (18)	403	42	90	2.1	9.6	4.5	8.7	1.0	0.17	1.1	~ 3000
12		Pic	42 (43)	15 (15)	20 (20)	21 (21)	98	12	23	1.9	8.2	4.3	11.1	1.8	0.25	4.8	~ 100
13		Trav	1 (1)	24 (34)	29 (41)	17 (24)	71	15	32	2.1	4.7	2.2	6.4	3.4	0.27	9.7	
Averages			**(16)**	**(24)**	**(38)**	**(22)**	**339**	**40**	**116**	**2.7**	**7.8**	**3.1**	**8.2**	**1.7**	**0.19**	**3.9**	
Overall avg.			**(11)**	**(24)**	**(39)**	**(26)**	**350**	**42**	**101**	**2.4**	**7.9**	**3.6**	**9.0**	**1.9**	**0.22**	**3.3**	
			Times in minutes. '(-)' is % of Total.														

Notes, by row:

3	Excludes tool-generated TCs (1 TC/59 SPs)
4	Excludes tool-generated TCs (121 TCs/121 SPs)
8	Excludes tool-generated TCs (93 TCs/93 SPs)
9	Excludes tool-generated TCs (40 TCs/40 SPs)

Table 4-3: Task development data: Round 2

	Who	Task Name	Builder Task Acq	Design	Coding	Test/ Debug	Total (Min)	TCs	SPs	SP/ TC	Min/ TC	Min/ SP	Sec/ TC Sym	Bugs/ Hour	Bugs/ TC	TC Sym/ Desc Wd	Desc Size (Words)
1		Hanoi	1 (2)	4 (9)	22 (47)	20 (43)	47	10	12	1.2	4.7	3.9	6.1	2.6	0.20	4.4	~ 100
2	S3	EMP	71 (8)	194 (22)	459 (51)	172 (19)	896	233	247	1.1	3.8	3.6	3.8	1.5	0.09	2.1	~ 5000
3		Predict	48 (7)	13 (2)	534 (73)	141 (19)	736	325	328	1.0	2.3	2.2	4.2	1.6	0.06	0.9	~ 10000
4		VT	50 (2)	456 (22)	1094 (53)	472 (23)	2072	301	739	2.5	6.9	2.8	5.9	1.2	0.14	1.3	~ 15000
Averages			(5)	(13)	(56)	(26)	938	217	332	1.4	4.4	3.1	5.0	1.7	0.12	2.2	
Overall avg.			(5)	(13)	(56)	(26)	938	217	332	1.4	4.4	3.1	5.0	1.7	0.12	2.2	
			Times in minutes. '(-)' is % of Total.														

Notes, by row:

4 Excludes tool-generated TCs (72 TCs / 91 SPs)

Table 4-4: Task development data: Round 3

4.3.1. Comments on the data

Tables 4-2 through 4-4 presented the raw experiment data. This section comments on the data and presents a variety of displays showing trends in the data. Section 5.2 presents additional displays that are relevant to analyzing TAQL's effectiveness.

I was not able to discern any trends or patterns in several of the data items: Soar productions (SPs) per TC, bugs per TC, and bugs per hour. There are variations in all of these, but the variations do not appear to be highly correlated with factors including task size, subject, experiment round, and task type (puzzle or expert system).

Figure 4-1 shows the percent of total development time spent in each major development activity, broken down by experiment round and averaged over each subject's tasks within a round. The major development activities are task understanding, design, coding, and testing/debugging.

Figures 4-2 and 4-3 display the relationship between description size and implementation size. Figure 4-2 plots the number of SPs in the implementation as a function of the number of words in the task description. The solid line was fitted using

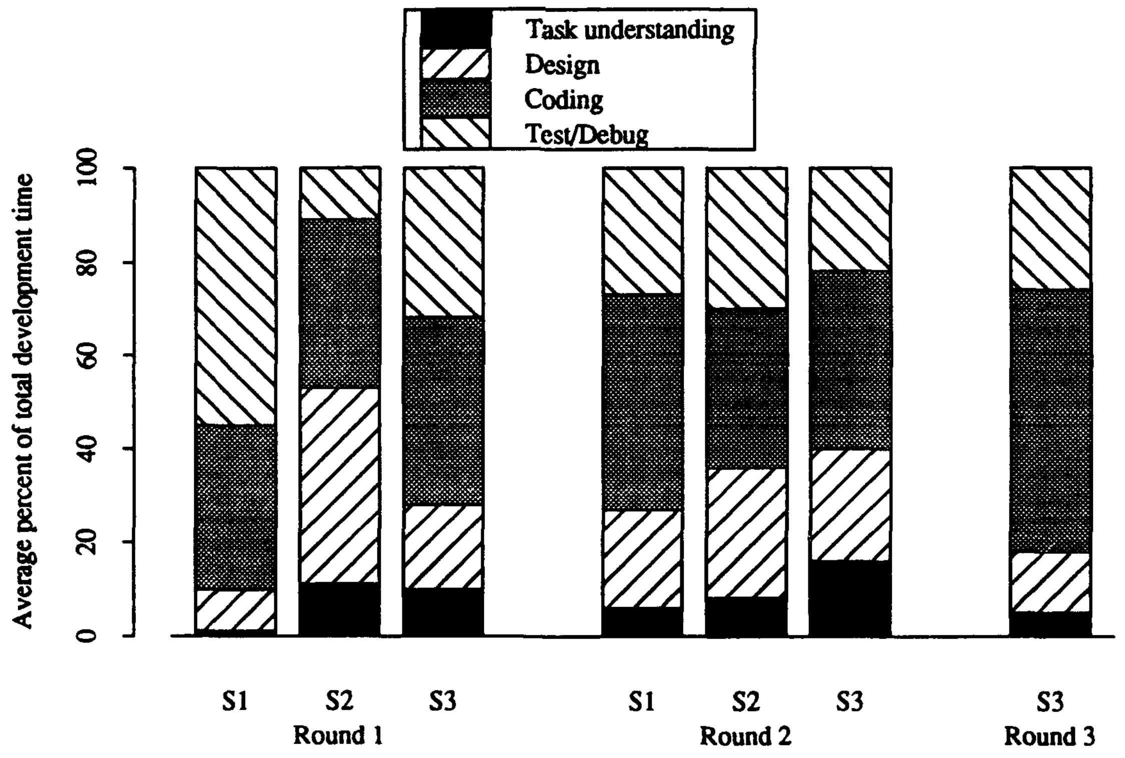


Figure 4-1: Average percent of development time spent in each development activity

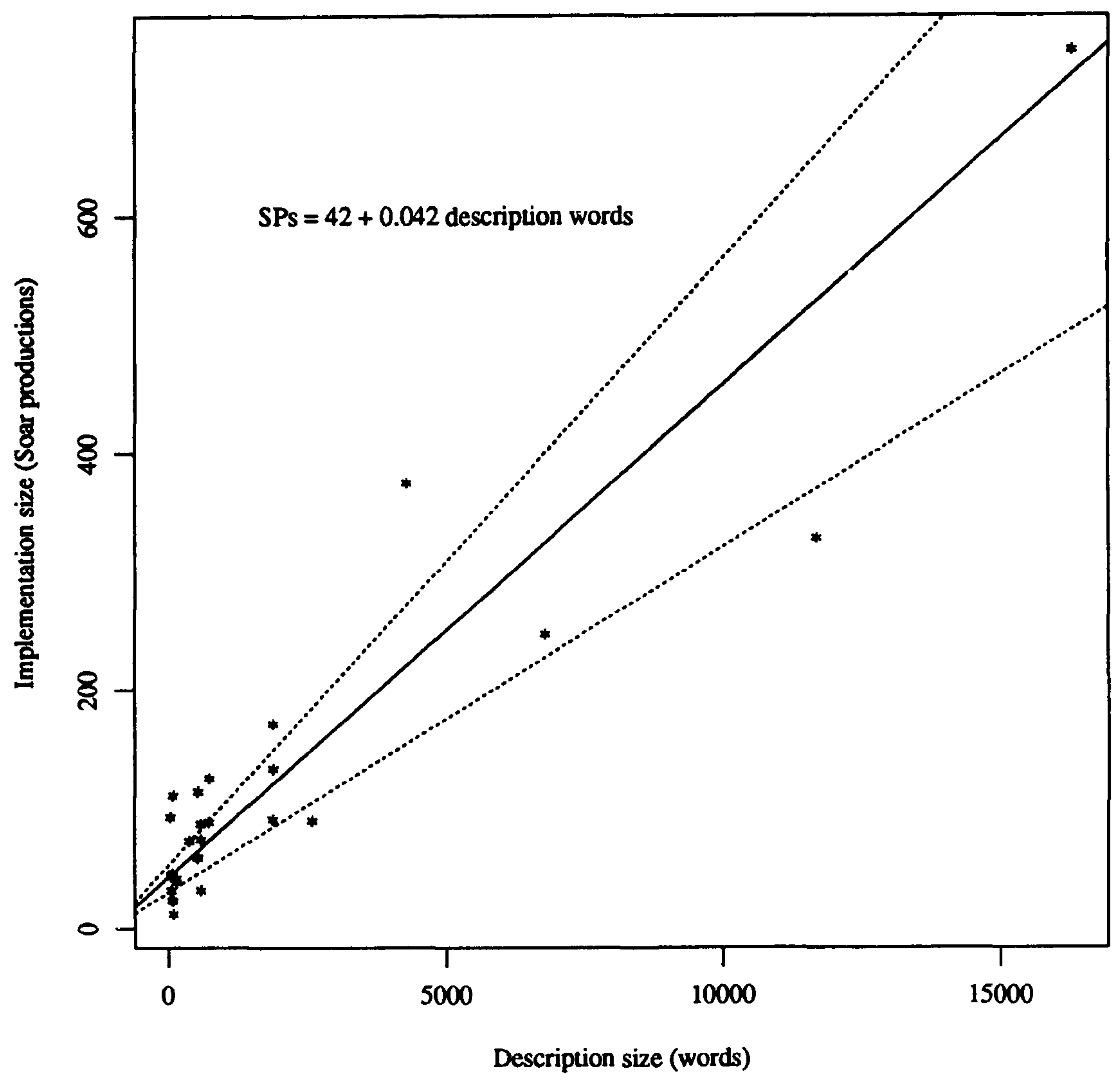


Figure 4-2: System size as a function of task description size

robust linear regression (Coleman et al., 1980), which is less sensitive to outliers and non-constant variance than least-squares regression. The dotted lines represent the upper and lower quartiles of the percent error from the regression line: half of the data falls between the dotted lines.

Figure 4-3 shows plots of the percent error of each actual system size from the size predicted by the regression line, using four different measures of system size: number of SPs, number of TCs, number of symbols in the TAQL program, and number of symbols in the Soar productions (SPs) compiled from the TAQL program. The line in the middle

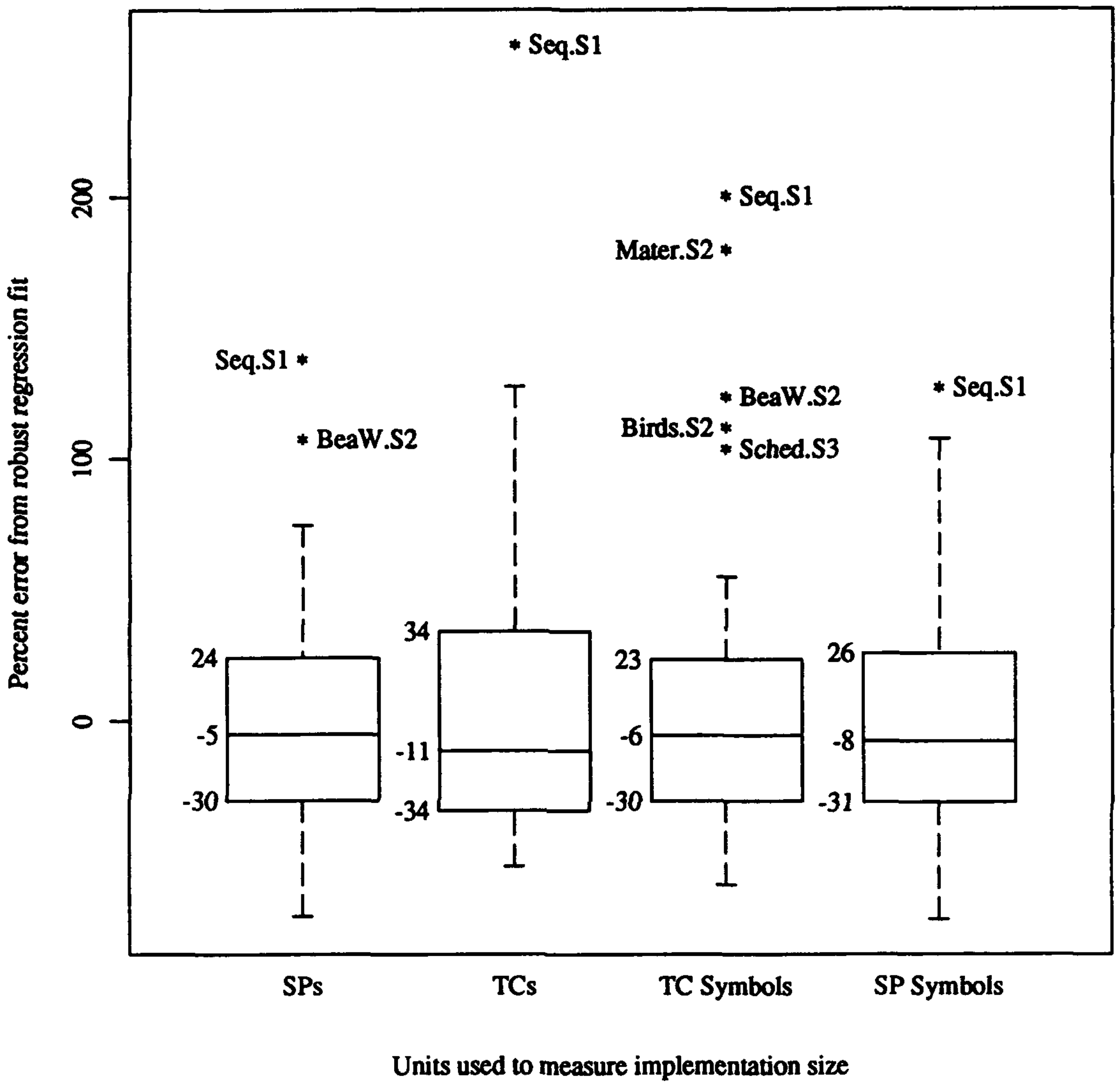


Figure 4-3: Percent error in system size regression, using four measures of system size

of each box is the median percent-error value, and the top and bottom lines of each box are the upper and lower quartiles.[9] The whiskers extending from the boxes extend to the farthest data point that lies within 1.5 times the height of the box of the nearest box side. Outliers are plotted separately and labeled. The plots show that number of SPs provides the best fit, followed by number of Soar production symbols, number of TAQL symbols and number of TCs. The regression fit appears to be a fairly good one using any of the measures, with half of the errors falling roughly within ± 30 percent.

[9]One-quarter of the data values lie between the median and a quartile.

Most of the data points in Figure 4-2 are clustered around the small descriptions sizes. Therefore, one might worry that the four largest tasks (Pipe.S3, EMP.S3, Predict.S3, and VT.S3) are exerting undue influence on the regression line. This is not true. Doing the regression with those four points discarded gives the same regression line: $42+0.042$ *description words*, rounded to the precision used here.

The puzzle task descriptions describe only the problem to be solved, while the expert-system task descriptions describe both the problem and its solution (in domain terms). Therefore, one would expect a puzzle task to require more TAQL code per description word than an expert system task. Figure 4-4 confirms this expectation, and also shows that there is more variation for puzzle tasks. This is not surprising, since puzzles by definition require an act of inventiveness that will vary from person to person.

Figure 4-4 also suggests that separate regressions for puzzle tasks and expert system tasks would fit the data better than the single regression shown in Figure 4-2. This may well be true, but the number of data points in each set is small enough and variable enough that the separate regressions are highly unstable (that is, the fit varies greatly when just a few data points are removed). The overall regression is much more stable.

4.3.2. Error data

The error data collected in the experiments was by far the strongest influence on how TAQL evolved between experiment rounds. The error data was studied to identify common classes of errors, and TAQL was changed to either make selected errors less likely to occur, or to make them easier to detect or fix. Section 4.4 describes the versions of TAQL used in the experiments and how it changed between rounds. This section classifies and analyzes the bugs that the subjects detected during the experiments.

Nearly all of the bugs that the subjects found in their experiment programs fall into four broad classes:

- *Typos*. A syntax error in the TAQL code. This does not include typos in user-defined items such as attribute names. Such typos are classified as *model errors*, defined below.
- *Design errors*. Gross errors in design that show up during implementation, by leading to poor, difficult, incorrect or impossible implementations without some redesign. An example is forgetting to design a problem space to perform some aspect of the task.
- *Implementation errors*. Errors where the programmer failed to correctly encode the problem space design they had in mind. An example is forgetting to include code to edit some aspect of the state that is specified in the system's operator-level design.
- *Model errors*. Model errors are a subclass of implementation errors, and are defined in more detail below.

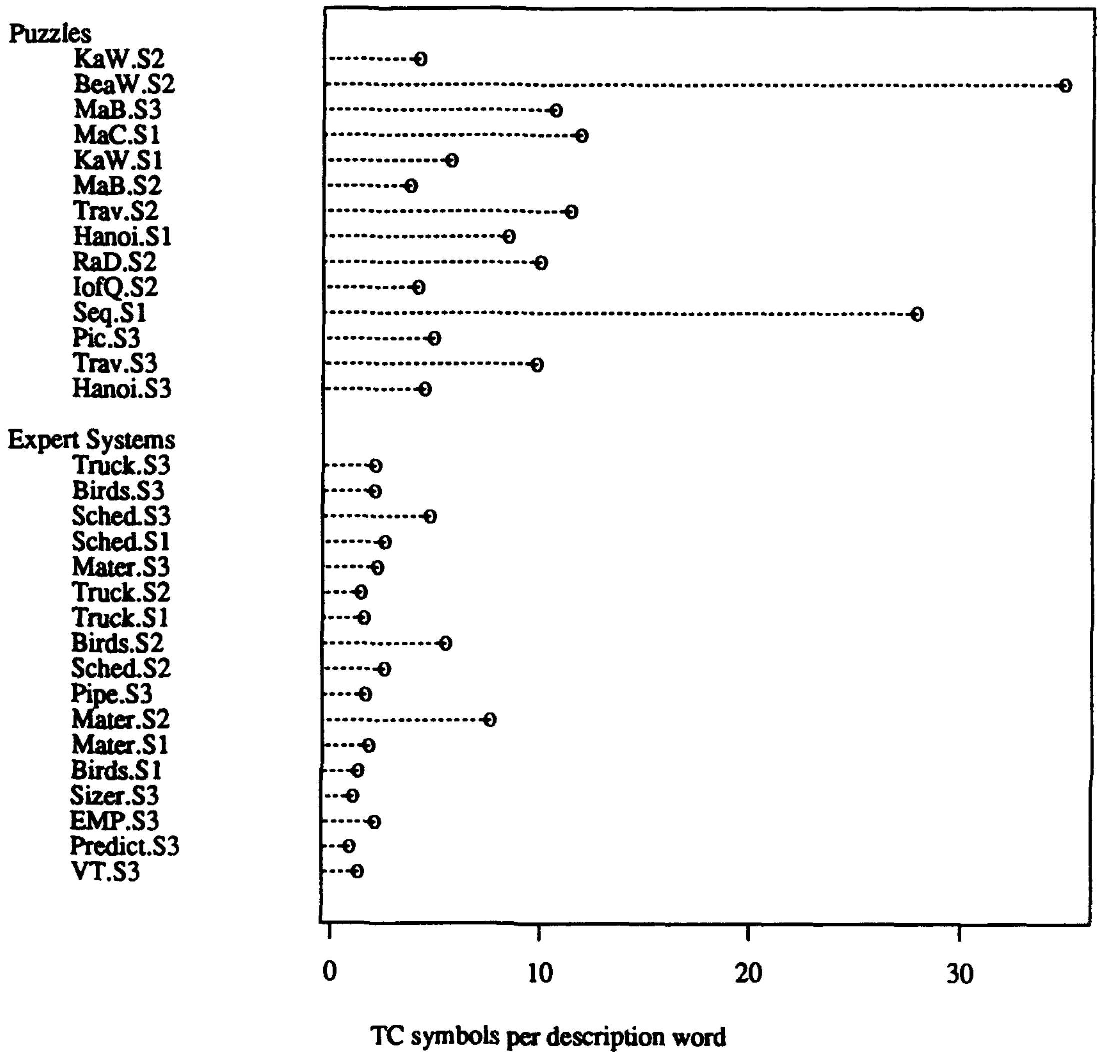


Figure 4-4: Task description density, grouped by task class

Model errors arise when the programmer has an abstract problem space design in mind (a *model* of the problem space), but in some way fails to properly transfer that design to TAQL. The distinction between model errors and implementation errors can be subtle. Model errors are implementation errors involving a failure to properly code some gross aspect of a problem space. For example, forgetting to propose an operator would be a model error, but proposing it under the wrong conditions would be a generic implementation error. Model errors subdivide into three classes:

1. *Space model errors.* The space model expresses the abstract structure of a problem space: what the operators are and whether the various PSCM knowledge roles are to be implemented directly or through problem solving in a subgoal. It does not contain details of data representations or of operator preconditions and effects. Example space model errors are forgetting to propose or apply an operator and omitting goal tests.

2. *Data model errors.* The data model expresses the structure of the data representations used in the task. It describes the classes of objects used, the attributes associated with each class, and the types of values that those attributes can take on. Example data model errors are misspelling attribute names, using attributes with the wrong object class, and omitting required operator parameters when an operator is proposed.

3. *Communication model errors.* The communications model expresses the flow of control and data within and between problem spaces. Within a problem space, it characterizes operator sequencing and what data objects are produced and used by various operators. Between problem spaces, it characterizes the role each space plays in problem solving, what data objects a space receives from its parent, and what data objects it returns as results. Example communication model errors are using the wrong attributes to pass back results and leaving out operator proposal conditions that constrain the flow of control.

The distinction between model errors and generic implementation errors is important, because model errors account for nearly half of all implementation errors and it is relatively easy to build tools to help programmers detect them. TAQL 3.1.4 provides several such tools, as described in Section 4.4.3.

Two classes of errors have been removed from all of the data presented in this document, and the time spent fixing them or working around them is not included in the reported total task development times (except that occurrences of these errors *are* listed in the individual bug data in Appendix E). These two classes are *text errors* and *Soar errors*. Text errors are mistakes in a task description. Soar errors are bugs in Soar itself. Nine text errors and eight Soar errors were detected during the experiments. Text errors and the time spent fixing them are excluded because the incorrect descriptions have all been fixed. Therefore, they would not recur if the experiments were repeated. Deducting the time to fix them will allow researchers using the corrected descriptions to validly compare their results to the results presented here. Soar errors were excluded for a similar reason. All of the Soar bugs that were encountered during the experiments were reported and, to the best of my knowledge, fixed. Therefore these errors would not recur if the experiments were repeated, and excluding the time spent fixing them will allow valid comparisons if the experiments are ever repeated with improved versions of TAQL. Furthermore, the experiments were intended to evaluate TAQL's quality, not the quality of the Soar implementation.

Subjects detected 279 bugs in their programs during the experiments, and spent 1999 minutes fixing bugs. Figure 4-5 shows the distribution of bug occurrences broken down by experiment round. Figure 4-6 shows the distribution of the time spent fixing bugs, broken down by experiment round. Each figure shows the distribution both in terms of bug class and time interval it took to fix the bug. The "other" errors referred to in the figures include errors such as misunderstanding the task description, misunderstanding TAQL's semantics, and procedural miscues in interacting with TAQL or Soar (for example, forgetting to re-load a modified TC during debugging,.

The figures show that implementation errors and its model-error subclass account for the majority of both bug occurrences and debugging time. They also show that while most of the bugs took less than ten minutes to fix, most of the debugging time went into

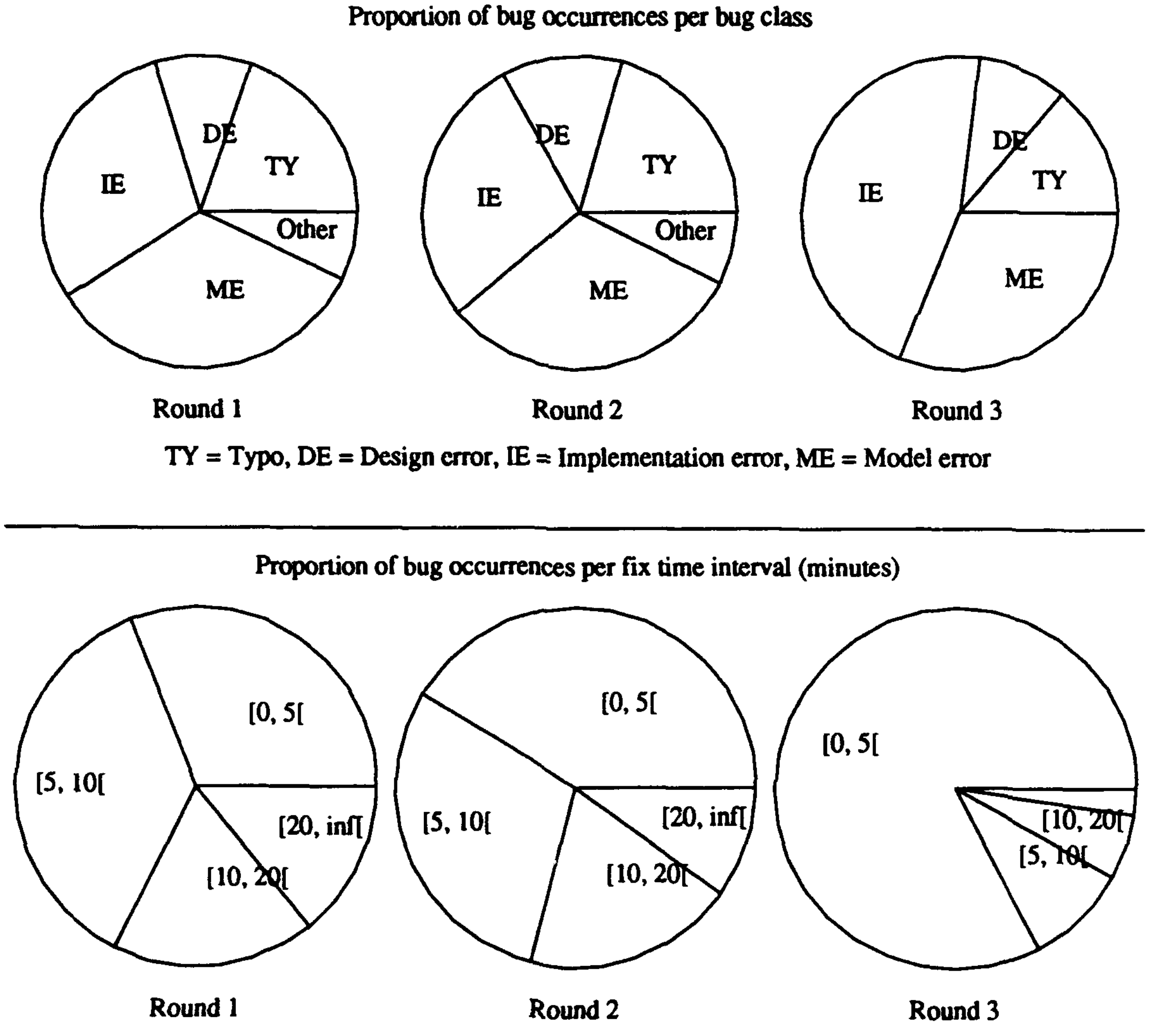


Figure 4-5: Bug occurrences, by experiment round

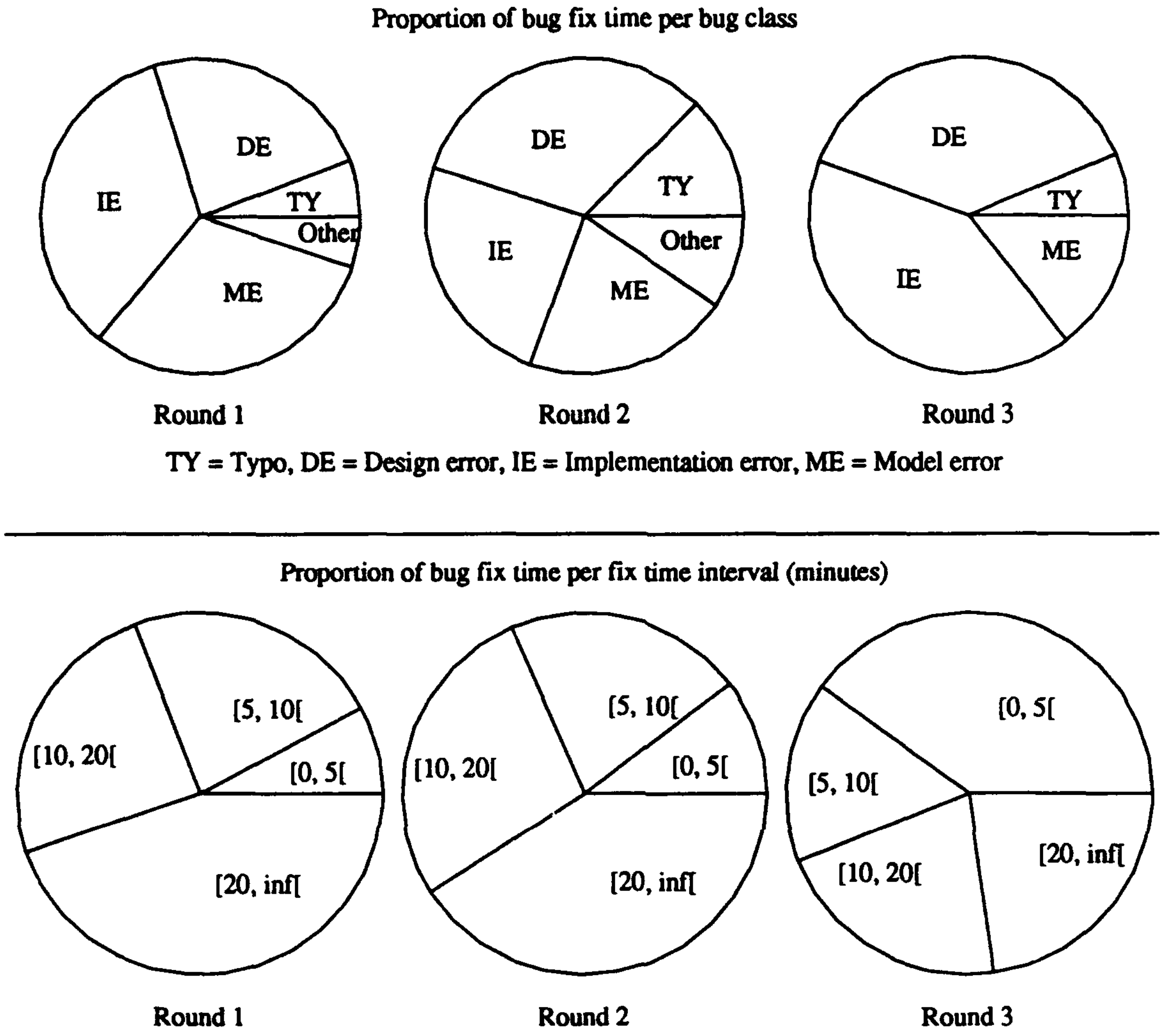


Figure 4-6: Bug fix time, by experiment round

fixing bugs that took more than ten minutes to fix. The shift to shorter fix times in Round 3 is probably largely due to the tools introduced in that round to make model errors easier to detect. See Section 5.3 for more details.

Finally, Figure 4-7 shows the average bug fix time, broken down by bug class and by experiment round within a class. The three largest bug classes (design errors, implementation errors, and model errors) all show steadily decreasing fix times. The class of "other" errors is so small (14 out of 279 errors) that its variations should not be considered significant. The decreasing fix times could be due to either learning on the part of the subjects or to improvements in TAQL. The data is inconclusive, but suggests the latter: only one subject (S1) exhibited learning effects (see Section 5.2.2), but Section 5.3 shows that at least some of the improvements to TAQL significantly reduced debugging time.

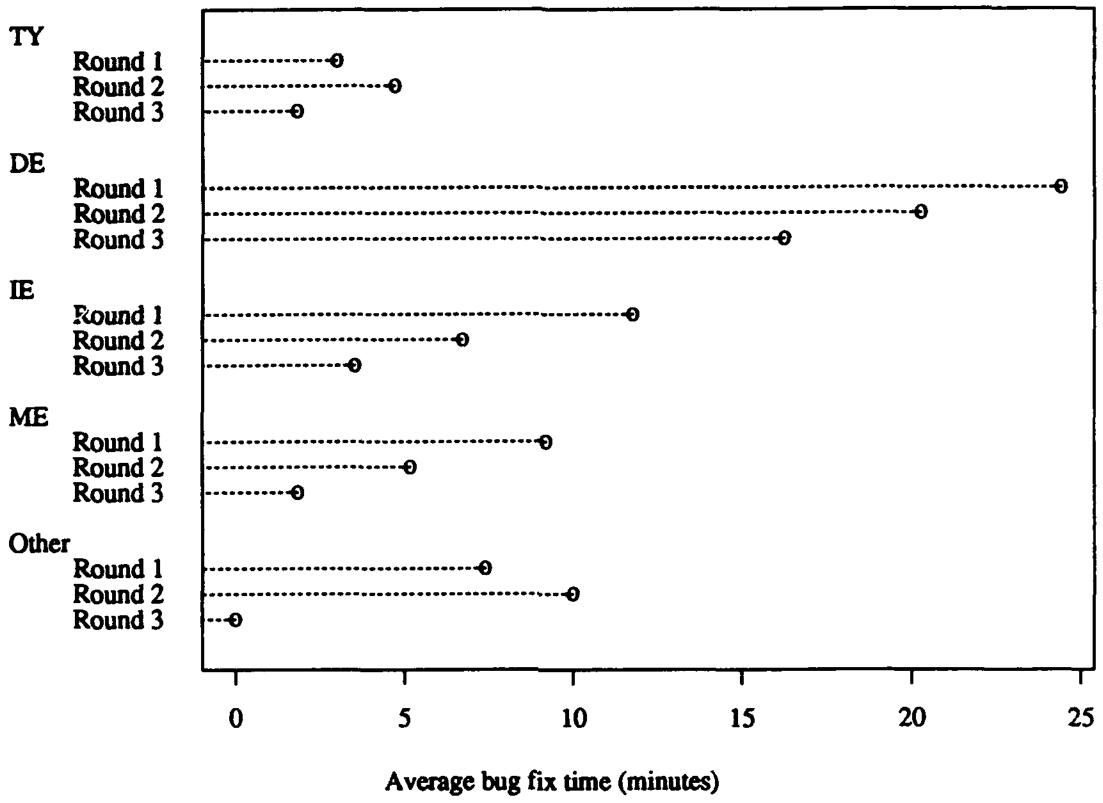


Figure 4-7: Average bug fix time, by bug class and round

Appendix E contains brief descriptions of all of the bugs that occurred during the experiments, broken down by experiment round, subject, and task.

4.4. The tools

Round 1 of the experiments used TAQL 3.1.0 (pre-release version), Round 2 used TAQL 3.1.3 (pre-release version), and Round 3 used TAQL 3.1.4 (pre-release and release versions). After each round, the bug data in the development logs was analyzed to see what types of mistakes people were making. Then, new TAQL features that would help developers avoid, diagnose, or fix those kinds of errors were designed. Some new features were added in response to user feedback (both the experiment subjects and other TAQL users).

Sections 4.4.1 through 4.4.3 describe the versions of TAQL used in the experiments and how and why TAQL changed between rounds. Section 4.4.4 describes TAQL modifications suggested by the final round of experiments.

4.4.1. Round 1: TAQL 3.1.0

TAQL 3.1.0 was the first version of TAQL to implement a complete PSCM (see Section 2.1). It had TAQL constructs corresponding to every PSCM knowledge role. It also supported problem-space relations of all types (prior versions of TAQL supported only operator-application relations and problem-space, state and operator selection relations). For example, in TAQL 3.1.0, it was possible to have one problem space propose the operators to be used in another problem space. All working memory objects had to be structured in terms of Soar's attribute value structures — there were no other data types.

TAQL 3.1.0's development environment was just Soar's development environment extended to apply to TAQL constructs as a whole rather than single productions. When a programmer gave a TC name as an argument to a Soar command, the Soar command was applied to all of the productions compiled from that TC.

4.4.2. Round 2: TAQL 3.1.3

TAQL 3.1.3 contained many extensions over TAQL 3.1.0. The most important were:

- Clearer and more powerful support for lookahead search and evaluation.
- A TC for expressing overall operator sequencing in a problem space.
- Support for four data types beyond Soar's attributes and values: lists, sets, expressions and text.
- A flexible, easy-to-use result-returning scheme for operator-implementation spaces.
- Assorted changes to existing TCs to improve consistency and flexibility.

The remainder of this section expands on the most important of these changes.

4.4.2.1. Support for evaluation and search

When Soar has insufficient directly-available knowledge to select among multiple operators (or problem spaces or states), its default action is to use the *selection space* in a subgoal (Laird, Congdon, Altmann and Swedlow, 1990). The selection space applies the *evaluate-object* operator to each of the competing operators in turn. The evaluate-object operator produces a numeric or symbolic (for example, *success* and *failure*) evaluation for the item it is evaluating. Knowledge in the selection space translates these evaluations into preferences, which the Soar architecture uses to make decisions. Whenever enough preferences have been created for Soar to be able to choose one of the competing operators, the selected operator is installed as the current operator and the selection space ceases to exist.

A programmer can supply domain-specific knowledge to tell Soar how to evaluate operators in the domain. This knowledge is in the form of operator-application knowledge for the evaluate-object operator. As always in Soar, the knowledge can either directly specify an evaluation, or compute it by doing further problem solving in subgoals.

In the absence of domain-specific evaluation knowledge, the selection space applies the evaluate-object operator by doing lookahead search. In a subgoal, it creates a copy of the problem-solving context in which the tied operators occurred, and installs the operator being evaluated. Problem solving proceeds in the subgoal until one of two things happens:

1. Domain-specific knowledge produces an evaluation for the current state in the lookahead context. In this case, the evaluation for the state becomes the evaluation for the operator being evaluated.
2. A final state is reached in the lookahead context. If the final state is a success state for the problem space, the operator being evaluated will be given a best preference, since it is on a path to the solution. If the final state is a failure state, the operator being evaluated will be given a worst preference, because it is not on a solution path.

During lookahead problem solving, other operator ties may arise. This leads to recursive applications of the selection space, and hence possibly to additional levels of lookahead search.

The selection space is extremely powerful. By adding small amounts of domain knowledge, it can be made to mimic the behavior of a wide variety of well-known search methods. Unfortunately, many programmers find the selection space rather confusing, something confirmed by several of the search-based tasks in Round 1 of the TAQL experiments. Soar productions that supply domain knowledge to the evaluation process must be tailored to fit in with the predefined productions in the selection space. The many details involved in coordinating with the predefined productions makes it difficult to focus on the domain knowledge to be added and the role it plays in the search.

After Round 1, Erik Altmann designed two new TAQL constructs (*evaluate-object* and *evaluation-properties*) for supplying domain knowledge to drive evaluation and lookahead search. The programmer supplies only the domain knowledge, and the TAQL compiler manages the details of integrating it with the selection-space productions.

The evaluate-object TC tells Soar what evaluations to produce and when to produce them. For example, suppose that we want to solve the monkey and bananas task (see Section 4.1) by using hill-climbing search, using the following knowledge to evaluate states:

- Assign an evaluation of 0 to states where the monkey and box are in different locations.

- Assign an evaluation of 1 to states where the monkey and box are in the same location, but are not at the bananas.
- Assign an evaluation of 2 to states where the monkey and box are both where the bananas are.

The evaluate-object TC expressing this knowledge looks like this:

```
(evaluate-object monkey-and-bananas*eo*lookahead
  :space monkey-and-bananas
  :what lookahead-state

  :numeric-value (0 :when ((state ^monkey-at <place>
                                  - ^box-at <place>)))

  :numeric-value (1 :when ((state ^monkey-at { <> bananas <place> }
                                  ^box-at <place>)))

  :numeric-value (2 :when ((state ^monkey-at bananas
                                  ^box-at bananas))))
```

The *:what* keyword specifies what kind of object the TC is giving evaluation knowledge for. In this example, the value is *lookahead-state*, indicating that the evaluation knowledge is for states produced during lookahead search. The *:numeric-value* keywords specify the evaluations to assign, and the *:when* keywords embedded in the value clauses specify the conditions under which that particular value should be assigned.

The second TC that supports evaluation is the *evaluation-properties TC*. It is used to declare any special properties of the evaluation function defined by the evaluate-object TCs, such as monotonicity. It is also used to define how states should be copied when creating a copy of the problem-solving context during lookahead search. TAQL provides a comprehensive set of defaults for all of these options, so that programmers often don't have to use an evaluation-properties TC. The TAQL manual provides complete details.

4.4.2.2. Localized expression of operator sequencing

The Soar architecture provides two ways to specify what order operators should apply in:

1. The conditions under which an operator is proposed can be designed so that the operator is only proposed when it should be selected.
2. Desirability preferences can be created among multiple proposed operators.

TAQL 3.1.0 provided only these two ways to specify operator control. In both of these approaches, control knowledge is distributed among several TCs. This can have great advantages: small bits of control knowledge can be dynamically assembled at run time to form a complete decision. This flexibility is very desirable in search-based problem spaces.

However, it is not so appropriate for more routine problem-solving where the operator sequencing is predetermined. It is easy to leave out parts of the intended control knowledge, and it is difficult to determine the control structure of a problem space by reading the program text. Such routine problem solving occurs as part of most expert system tasks. Many of the bugs in the small expert system tasks encoded in Round 1 of the experiments could have been avoided if there had been a concise way to specify operator sequencing for routine subtasks. This was particularly true for the bird classification and shipment scheduling assistant tasks.

After Round 1, a new TC (the *operator-control* TC) was designed for locally specifying operator sequencing for a problem space. An important design constraint was that it should coexist peacefully with Soar's distributed control mechanisms: it could not simply impose a purely procedural language on top of Soar.

The solution was to specify control in terms of templates that specify legal operator sequences. The primitive components in the templates match against operators. The primitive components can be composed into disjunctions, sequences, and loops. I will refer to these templates as *control templates* throughout this section.

For example, suppose that the operators in a problem space must apply in one of two sequences: operator A followed by operator B, or operator B followed by operator C (A, B, and C here are operator names). The TAQL control template to specify this ordering would be

```
(:or (:seq A B)
     (:seq B C))
```

At run time, a proposed operator will be rejected if it would create an operator sequence that is inconsistent with an active control template. For example, suppose that the preceding control template is active, and that operator A is the first operator that was selected in the space. If operator C is one of the operators proposed after A applies, it will be rejected because the template specifies that only B can follow A.

In this example, the primitive components in the control template are simply operator names: A, B, and C. In general, they can be any TAQL condition that matches against an operator object. For example, suppose that in a job-shop scheduling system, operators are tagged with the effect they produce (for instance. grind, polish, perforate, and so forth). In addition, suppose that operators may sometimes be tagged with a priority attribute. The following control template would then specify that all operators that either grind, perforate, or are urgent should be performed first, followed by any remaining operators that polish.

```
(:seq (:loop :ind
         (:or (:any-name ^effect grind)
              (:any-name ^effect perforate)
              (:any-name ^priority urgent)))
      (:loop :ind (:any-name ^effect polish)))
```

The lists in this example that begin with *:any-name* are templates for individual operators. The :any-name keyword specifies that the operator name is irrelevant. The :ind (*indifferent*) keyword specifies that if there is more than one operator that could be selected on the next iteration of loop, it does not matter which one is selected. If the programmer wants finer control of which operator is selected at each iteration, there are two options. First, the control template could be made more complex to reflect the additional control constraints. Second, the additional control could be specified with TAQL *prefer* and *compare* TCs or determined by problem-solving in a subgoal. For example, the programmer might want to specify that subject to the ordering imposed by the above control template, hot objects should be processed after cold objects. This could be accomplished by adding the following compare TC:

```
(compare job-shop*c*cold-first
  :space job-shop
  :op1 (:any-name ^temperature cold)
  :op2 (:any-name ^temperature hot)
  :relation better)
```

As this example indicates, control templates can be mixed freely with the distributed control specifications provided by prefer and compare TCs. Further, since the primitive components of control templates are operator patterns rather than specific operators, the control templates are not simply an algorithmic control language in the traditional sense. They make it easy to under-specify the legal operators sequences. This is a very desirable feature in a language that is intended to express intelligent action, which is rarely easy to specify in purely algorithmic terms.

Another feature of operator-control templates is that programmers can specify conditions under which a control template is to be activated and deactivated. Thus, problem-solving may be under the control of a control template only at some points. The control templates provide many capabilities beyond the ones described here. For example, a control template can be interrupted and later resumed, and the looping constructs provide for conditional looping and non-local loop exits. The TAQL manual provides complete details.

4.4.2.3. Data types

TAQL has a rudimentary data-type facility. It provides four data types beyond the basic attribute/value data type: *sets*, *lists*, *expressions*, and *text*. Currently, programmers cannot define new data types, although the expression data type is extensible (the TAQL manual provides complete details).

Each data type provides a specialized notation for denoting literal objects of that type. For example, the notation for the list data type is the parenthesized notation used in Lisp. Each data type also provides a set of operations on objects of that type. These operations are either *operators*, *data macros*, or *directives*. Operators are just ordinary Soar/TAQL operators. For example, the expression data type provides an *eval-expr* operator that the programmer can select to evaluate a specified expression. Data macros and directives, on the other hand, can be used within a TC's conditions or actions to succinctly match, modify, or create objects of some data type.

The following apply-operator TC uses data macros from the list and text data types to implement an operator named *add-title-to-name*. The state in the *make-titled-names* problem space initially contains two list objects, *titles* and *untitled-names*. Each is a list of strings. *Titles* is a list of titles such as "Dr.", "Mr.", and "Ms." *Untitled-names* is a corresponding list of names. The *add-title-to-name* operator takes the first item from each of these lists, concatenates the two items into a single text object, and adds the result to a third list, *titled-names*. The data macros involved are:

- *Car*, *cdr*, and *cons* from the list data type. These are defined as in Lisp.
- *Text* and *concatenate* from the text data type. *Text* coerces symbols, numbers, and strings to an appropriate object of the text data type. *Concatenate* concatenates an arbitrary number of text objects into a single text object.

```
(apply-operator make-titled-names*ao*add-title-to-name
  :space make-titled-names
  :op add-title-to-name

  (edit :what state
        :when ((state ^titles <titles>
                      ^untitled-names <names>))

        :replace (titled-names <titled-names>
                   :by (cons
                         (concatenate
                           (text (car <titles>))
                           (text (car <names>)))
                        <titled-names>))

        :replace (titles :by (cdr <titles>))

        :replace (untitled-names :by (cdr <names>))))
```

TAQL compiles all data types into attribute/value representations. Currently, TAQL systems are traced at the Soar level rather than at the TAQL level — the programmer sees productions firing rather than TCs applying, for example. This is also true of data types. At run time, the programmer will see the attribute/value structures the data types are compiled into, rather than the more concise notations used in the data macros and directives in the source code. This can make traces difficult to read, and TAQL should be changed to display traces using the same notations that appear in the programmer's source code.[10]

These data types were added to TAQL to facilitate the representation mapping function (see Section 2.4) for tasks that made natural use of these data types. None of the Round 1 and 2 tasks did, and so the data for those rounds did not suggest that data types would be useful. Some of the Round 3 tasks used the data types extensively.

4.4.3. Round 3: TAQL 3.1.4

The core TAQL constructs barely changed between Round 2 (TAQL 3.1.3) and Round 3 (TAQL 3.1.4) — in fact, TAQL 3.1.4 was upward compatible with TAQL 3.1.3. Nearly all of the changes between these two version involved extending TAQL's programming environment. Data from the first two rounds suggested that the existing environment was barely adequate for even small expert systems, and the third round was to contain several large expert-system tasks.

Round 3 used two different versions of TAQL 3.1.4. I expected that when TAQL was first used on a large expert system task, a number of possible TAQL improvements would become apparent. Since there was no time for another full round of development and experiments, I decided to make any small adjustments that seemed desirable after the first large expert-system experiment (the EMP task), and use the modified TAQL for the remainder of the final round.

The most important new features in the first version of TAQL 3.1.4[11] were

- A more concise syntax for describing working-memory objects.
- Gnu-emacs TAQL and Soar modes that include a structured editor for TAQL constructs (Ritter, 1991a; Ritter & Hucka, 1991). The editor helps TAQL programmers fill in syntactic TAQL construct templates. It is menu-based,

[10]Source-level tracing is difficult to implement in the face of learning. Soar's learning mechanism operates at the production level, and it is not clear how to re-express the learned productions in terms of TAQL.

[11]The TAQL 3.1.4 release notes (Yost and Altmann, 1991b) provide a complete description of the changes from TAQL 3.1.3.

provides for auto-completion of symbols (such as attribute and variable names), and has extensive on-line help facilities. Routine use of the editor would eliminate most syntax errors. Also, the menus and templates remind programmers what TCs and keywords are available.

- A way to name segments of program text, plus the extension of interface commands to operate on entire code segments instead of individual TCs.
- Tools for declaring, inferring and printing the structure of problem spaces and for checking that the actual problem space structure is consistent with its declaration. Also, tools for inferring and printing the relations among problem spaces.
- Tools for declaring, inferring and printing the structure of data objects used in a TAQL program.
- A tool to find all TCs that contain conditions or actions matching specified templates.

The most important features added after doing the EMP task were

- A way to easily detect multiple TCs that were accidentally given the same name. TAQL does not normally warn the programmer when loading a TC that has the same name as an existing TC. Instead, it excises the old version of the TC, and installs the new one. This is the right action to take during debugging, when the programmer wants to load a modified TC to fix a bug. However, it makes it hard to detect different TCs that were accidentally given the same name. TAQL 3.1.4 has an option that tells it to warn about duplicate TC names.
- Improved support for loading operator libraries. Formerly, programmers had to use Lisp's load function to load TAQL's operator libraries (which come with TAQL's data types — see Section 4.4.2.3). TAQL 3.1.4 provided a new command, *use-taql-library*, which takes the name of a data type and loads the appropriate library. The details of where the library is installed are hidden from the programmer. Also, use-taql-library keeps track of which libraries are loaded, so that they will not be loaded more than once.
- The ability to use raw Soar actions in operator applications and object elaborations. TAQL provides a simple high-level language for making changes to objects during operator applications. Changes are specified functionally rather than in terms of raw Soar preferences. For example, TAQL has keywords for creating new attributes, replacing the value of an attribute, copying attributes, and so forth. The TAQL compiler converts these functional specifications into Soar preferences that have the same effect. The functional form has clear semantics, but there are drawbacks. It is tailored towards describing how to change single objects in isolation. It can be difficult to specify coordinated changes to a set of objects. To remedy this, TAQL 3.1.4 allows programmers to specify an operator's actions directly in terms of raw Soar actions.

The remainder of this sections expands on the most important of these changes.

4.4.3.1. Program segments

Prior to TAQL 3.1.4, TAQL's interface commands all operated at the level of individual TCs. This is tolerable for small systems, but unwieldy for large systems. Programmers often want to operate on entire program sections at once, for example to print or trace all of the TCs in a particular problem space.

TAQL 3.1.4 provides a way to group together related TCs and declarations into subsections of a TAQL program. These groupings are called *segments*. Segments have programmer-assigned names and may be comprised of disjoint program sections. All TAQL interface commands that formerly accepted TC names as arguments also accept segment names in TAQL 3.1.4.

Two new commands, *begin-segment* and *end-segment*, are used to define a segment. A third command, *segment-group*, assigns a name to a group of segments or segment groups (thus permitting hierarchical segments).

4.4.3.2. Problem space and data structure declarations

Many of the bugs in Rounds 1 and 2 can be classed as *model errors* (see Section 4.3.2). As discussed in Section 4.3.2, model errors subdivide into space, data, and communication model errors. This section describes a set of tools that were designed after Round 2 to make it easier to detect model errors. Section 5.3 analyzes the effectiveness of these tools.

The operator-control TC in TAQL 3.1.3 (see Section 4.4.2.2) was designed to reduce communication model errors by concisely specifying all operator sequencing knowledge for a problem space in a single location. TAQL 3.1.4 includes similar facilities for problem-space and data models. It allows programmers to abstractly declare the structure of problem spaces and data objects. As TAQL loads TCs, it builds internal models of the problem spaces and data objects actually expressed by the TCs. By comparing these inferred models to the programmer's declarations, TAQL is able to detect a wide variety of space and data model errors. I will refer to the models described by declarations as *declared models*, and to the models built while loading TCs as *inferred models*.

The *defspace* command in TAQL 3.1.4 forms the declared models of a problem space. It has keywords corresponding to each of PSCM's knowledge roles, such as initial-state proposal and goal testing. The value of each keyword declares whether the knowledge for that role is implemented directly by TCs, by problem solving in a subspace, or by some combinations of the two. The programmer can also declare that the problem space does not make use of knowledge of that type. Defspace also accepts a list of the names of the space's operators and abstract information about how each operator is proposed and applied.

An example defspace command appears below. It is taken from the code for the elevator-configuration experiment task, and is for a problem space named *VT*. In the VT space, initial-state proposal, state entailment, goal testing, result returning, and operator selection are all done directly by TCs in the space. There is no evaluation knowledge. The VT space has three operators: *ask-questions*, *fix-violation*, and *report-results*. All three operators are proposed directly by TCs in the space. The ask-questions operator is applied directly, the fix-violation operator is applied partially by TCs in the VT space and partially in a subspace, and the report-results operator is applied entirely in a subspace.

```
(defspace VT
  :initial-state-proposal       direct
  :entailment                   direct
  :goal-testing                 direct
  :result-returning             direct
  :selection                    direct
  :evaluation                   none

  :default-operator-proposal    direct
  :default-operator-application (subspace direct)
  :operators (
     (ask-questions :apply direct)
     fix-violation
     (report-results :apply subspace)
   ))
```

TAQL 3.1.4 provides several commands for declaring data structures and operators. I will describe only one of them in detail. The TAQL 3.1.4 release notes (Yost and Altmann, 1991b) provide complete details.

The *defobject* command declares the attribute/value structure of a class of data objects. The programmer lists the attributes that can be used for objects of the given type, and the types of values the attribute accepts. Defobject declarations can be tagged as global or local declarations, with local declarations being specific to the program segment (see Section 4.4.3.1) they appear in. Also, the programmer can indicate that not all of the legal attributes for a type have been declared. This limits the number of errors TAQL can detect, but it is often useful during initial development, as will be described more fully below.

Below is an example of the defobject commands that define a simple linked-list data type. First, a *defenumeration* command defines a primitive type named *null*, which consists of the single symbol *nil*. Then, a defobject command defines the *list* object type. List objects have two attributes, *car* and *cdr*. The value of *car* can be of any type, but the value of *cdr* must be of either type *list* or *null*.

```
(defenumeration null
  nil)

(defobject list
  :global yes
  :all-info-declared yes

  car
  (cdr :type (list null)))
```

TAQL 3.1.4 includes other type declaration commands for merging new attribute declarations into an existing object declaration and importing local type declarations from another segment. These commands are useful in programs that use the same or similar types in several program segments (TAQL programmers often define program segments so that each segment contains all of the code for a single problem space). There are also commands for defining primitive data types (such as numbers, symbols, and subsets thereof), and for defining operators and their parameters.

TAQL provides interface commands for printing its inferred space and data models. These commands print the models in the form of declaration commands that programmers can insert in their programs. Thus programmers can write their programs initially with only partial declaration (or no declarations at all), load the programs so that TAQL can build its inferred models, and finally print out the inferred models and insert them as declarations. This lets programmers easily get the benefit of improved error checking that declarations allow without having to spend a lot of time at the outset writing declarations. However, TAQL programmers have reported that they find it better to write complete declarations at the outset. This enables extra error checking even for the initial version of a system. More importantly, writing the declarations in advance clarifies the system's abstract structure in the programmer's mind.

As TAQL loads a program, it also builds a model of how problem spaces relate to one another. It bases this model on the program's propose-space TCs, which define the functions of the system's problem spaces. Programmers can print out this model (part of the communication model) to get a concise picture of how their spaces fit together and what operators are in each space. This makes it easier to detect errors such as omitting propose space TCs or forgetting to implement some problem spaces.

4.4.4. TAQL changes suggested by Round 3

The results of Round 3 suggest several ways to improve TAQL further. All of these involve improvements in TAQL's programming environment rather than in the core structure of TAQL itself.

- Make use of information about the relationships among problem spaces when checking problem spaces and data structures against their declarations.

This would let TAQL do better error checking on the attributes and values passed between to problem spaces.

- Allow programmers to declare more easily that an attribute takes on values from an enumerated set. It is currently possible to give such declarations, but it is unwieldy enough that programmers typically don't bother to do it for their initial prototypes. In particular, the declarations produced by printing an inferred data model should constrain primitive attributes to take on only the values that are actually used with the attribute in the program.

- Allow programmers to declare more information about data structures. Currently, programmers can only declare which attribute a structure has and what the legal values for those attributes are.

  For example, when creating multiple values for an attribute in Soar, the values must be given *parallel preferences*. A common bug is to forget to create these preferences. If programmers could declare which attributes could have multiple values, TAQL could automatically create the appropriate preferences.

- Perform some level of data-flow analysis. Ideally, this would let TAQL produce displays of how working-memory objects flow among operators and problem spaces. But even a very simple version that kept track of which attributes were created and used would be very useful. For example, in the elevator configuration task it was difficult to find formulas that used values that were never computed anywhere. Even very simple data-flow analysis could have detected such errors.

- Provide a way to checkpoint the runtime state of a system being tested and restart from that state later. Currently, once Soar has run past the point at which a bug occurs, the only way to get back is to run it again from the beginning (or to checkpoint the entire Lisp image on disk, which takes many megabytes). This can make debugging large systems very time consuming.

- Provide a way to easily monitor the changes operators make during problem solving. Currently, all of Soar's trace levels that print working memory changes print far too much information to be used routinely.

A graphic interface would make testing and debugging much easier. Soar has enough tracing and debugging commands that most programmers are only familiar with a small subset. A menu-based interface would be able to organize these commands and help programmers find appropriate ones. Also, it can be difficult to sort though the linear textual traces Soar currently provides. Graphic traces could display information more abstractly and still let the programmer selectively display parts in more detail. A Soar/TAQL interface (Ritter, 1991b; Ritter, 1991c) that includes these features is being developed, and an initial version is now being used by a small set of users.

A graphic interface could also contain facilities for graphic programming. For example, programmers could use a graphic editor to build problem-space diagrams of the kind displayed by the graphic tracer (Figure 2-1 on page 9 shows an example hand-produced problem-space diagram). TCs defining problem-space relations could then be compiled directly from the diagrams. Similarly, search-control constructs could be compiled from operator flow graphs.

Chapter 5
Evaluation

My thesis is that one can apply the strategies that have been used to design method-based tools to a broadly-applicable weak method and still obtain a highly-effective tool. This chapter evaluates TAQL on the dimensions of scope and effectiveness. It shows that TAQL has broad applicability and yet is able to outperform method-based tools.

5.1. TAQL's scope

TAQL's scope is apparent from the range of tasks used in the experiments (see Section 4.1). These tasks divide broadly into expert system tasks and knowledge-lean puzzle tasks. The expert-system tasks include both analytic and constructive tasks:

- *Analytic tasks*: material-handling equipment selection, bird classification, EMP hardness evaluation, and construction-time prediction.
- *Constructive tasks*: job-shop scheduling, shipment scheduling assistant, computer-system sizing, oil-pipeline scheduling, EMP hardness evaluation, and elevator configuration.

The EMP hardness evaluation task is both analytic and constructive: it must first evaluate an electromechanical design, and then generate a report describing the system and its evaluation.

Many members of the Soar community use TAQL to build the systems involved in their research. The tasks include:

- Railroad switching: manipulate train cars and engines on tracks with sidings. The system uses a wide variety of abstraction and problem solving methods, and exhibits many different kinds of learning.
- Dispatching messages for a large organization, making use of an external database describing the organization, its members, and the available communication paths. As part of its problem solving, the system must decide how it will use the database and how to formulate its queries in SQL.
- Formulating mathematical models for operations research problems. Given a domain-oriented problem description, the system formulates the problem as a linear or mixed integer program.

- Modeling the behavior of a computer user interacting with an on-line information browser.
- Modeling the behavior of people involved in dyadic multi-issue negotiations.
- Modeling how people acquire strategies for solving the Towers of Hanoi problem.
- Modeling the acquisition of number conservation knowledge in children.
- Modeling human behavior on highly interactive tasks.
- Modeling how people use and learn to use automated teller machines.
- Modeling the behavior of the NASA Test Director during a space shuttle launch.
- Selecting engineering design tools for high-rise buildings.
- Critiquing floor plans for constructibility problems.
- Scheduling manufacturing production for windshields.

These systems illustrate some TAQL capabilities that were not apparent in the experiment tasks. Most of these systems use Soar's learning mechanism, chunking. Many of them interact in sophisticated ways with the external environment (either the real world, simulated worlds, or external software systems). They also demonstrate that TAQL can express psychological models.

In the TAQL experiments, subjects only developed systems to the point of an initial prototype. That many people are using TAQL voluntarily for their own research systems shows that TAQL is not just useful for prototyping. All of these systems have existed for months or years, and nearly all of them are under continuing development. They include some of the largest Soar systems in existence, with some having over 1000 productions. The message-dispatching system starts out with over 1800 productions and has grown to nearly 12000 productions due to learning over the course of extensive operation (Doorenbos, Tambe and Newell, 1992).

5.2. TAQL's effectiveness

TAQL has proven to be a highly-effective expert-system development tool. Section 5.2.1 shows that TAQL outperforms two well-known method-based tools, KNACK and SALT, on tasks that are very well-suited to those tools. Section 5.2.2 presents additional evidence of TAQL's effectiveness. The data from the TAQL experiments indicates that TAQL is equally effective for small and large tasks and that novice users can reach expert-level performance very rapidly.

5.2.1. Comparison to KNACK and SALT

As discussed in Chapter 3, there is very little data available on the effectiveness of other expert-system development tools. This makes effectiveness comparisons difficult. Fortunately, for two of the expert-system tasks in the TAQL experiments, there is data on the implementation of those tasks using other tools. These are the EMP hardness evaluation (EMP) and elevator configuration (VT) tasks, originally implemented using the KNACK (Klinker, 1988) and SALT (Marcus, 1988) knowledge acquisition tools, respectively.

Both KNACK and SALT are method-based tools, and so should be highly effective for tasks their built-in methods are appropriate for. Since it was these two tasks that largely drove the initial development of their respective knowledge acquisition tools, it is reasonable to assume that the built-in methods match these two tasks quite well.

The KNACK and SALT developers report the development effort that went i[illegible] these two tasks (Klinker et al., 1989; Marcus et al., 1985). Klinker and colleagues refer [illegible]heir system for performing the EMP task as the *DPR WRINGER*. Marcus and colleagues refer to their elevator-configuration system as *VT*. For clarity, I will refer to these systems as EMP/KNACK and VT/SALT, and to the TAQL versions as EMP/TAQL and VT/TAQL.

Figure 5-1 depicts the development effort that went into these systems. Klinker reports that EMP/KNACK took 13.5 days to implement, which I will convert to 81 hours.[12] EMP/TAQL took only 15 hours. Marcus reports that VT/SALT took 46 hours to implement. VT/TAQL took only 35 hours.

The experimental situations were quite similar in all cases. All of the systems were implemented by people very familiar with the tool they were using and at least somewhat familiar with the domain. The primary TAQL developer (S3) implemented EMP/TAQL and VT/TAQL, and also wrote the domain-oriented descriptions for those tasks over a year earlier (after which he did not look at them again until he began their implementations in the experiments). The versions of EMP/KNACK and VT/SALT described here were both reimplementations of earlier versions of the systems, implying that the developers were not only familiar with the tool and the domain but also familiar with the prior implementations. The person who implemented EMP/KNACK was a member of the KNACK group but was not a KNACK developer. VT/SALT was implemented in part by the primary SALT developer and in part by the primary VT

[12]I convert from days to hours assuming six hours a day were spent working on the system. Klinker (personal communication) says that the 13.5 days reported in their paper did not exclude breaks and distractions. The six hours per day is my estimate, and is intended to be conservative.

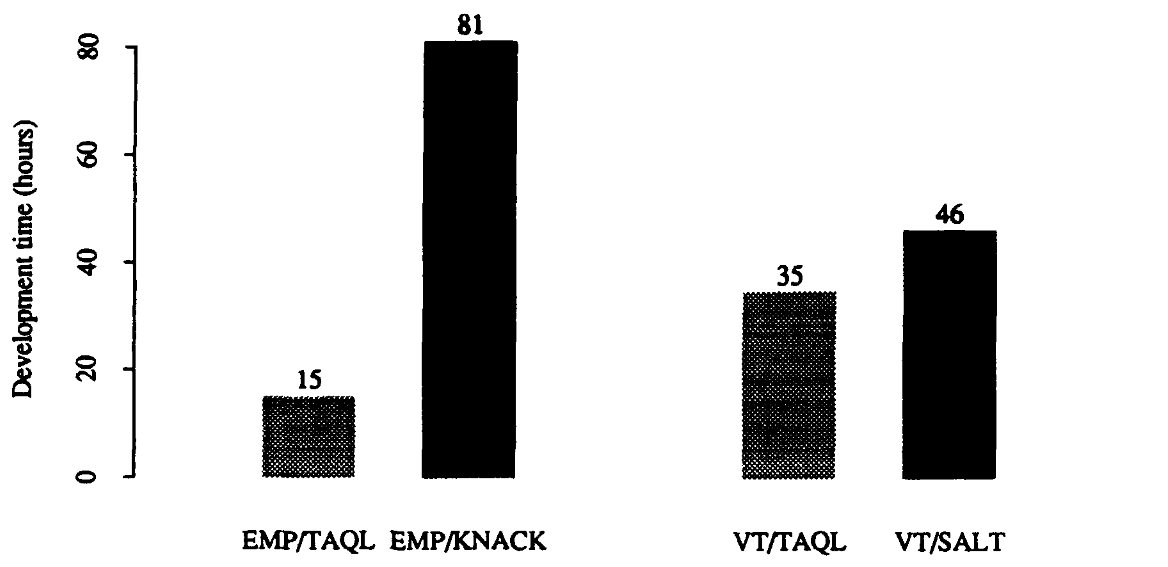


Figure 5-1: Development times for EMP and VT using TAQL, KNACK, and SALT

domain expert. (The SALT developer spent 11 hours entering about half of the knowledge, using the earlier implementation of VT as a knowledge source. The domain expert spent another 35 hours checking and refining that knowledge and entering the remaining knowledge.)

Therefore, in all cases, the task developers were at least somewhat familiar with the task beforehand, and were also very familiar with the development tool they used. If anything, it seems that these factors would mitigate in favor of the versions developed using KNACK and SALT, since their developers had been immersed in the task and the tool for months or years prior to building the versions of the systems reported here.

At the outset, I did not expect TAQL to outperform method-based tools. I expected only that it would surpass them in scope and be more effective than other general-purpose tools. Briefly, I attribute TAQL's effectiveness to a combination of the principled way in which TAQL was designed and PSCM's simplicity and flexibility. Section 6.1 discusses this in more detail.

5.2.2. Other evidence of TAQL's effectiveness

Figure 5-2 plots the encoding rate for each subject/task combination. The encoding rate is the number of minutes per Soar production, computed by dividing the total task-development time by the number of Soar productions the final TAQL system compiled into. The figure groups the data by subject, and within subject the rows are in the order

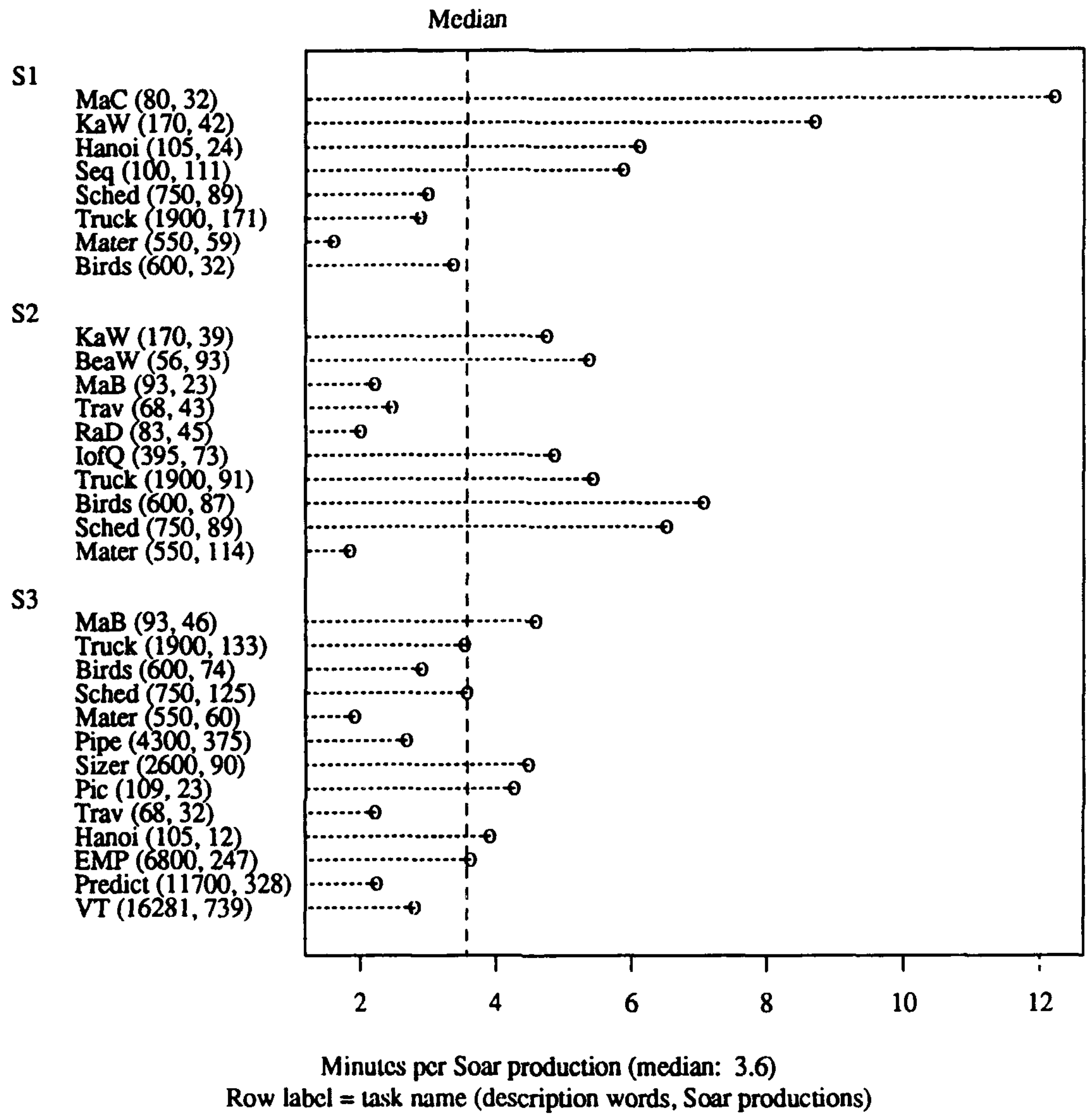


Figure 5-2: Encoding rates, by subject. Ordered chronologically within subject.

that the subject developed the tasks. The numbers in parentheses after each task name in the figure are the number of words in the task description and the number of productions compiled from the final TAQL system, respectively.

The encoding rates are quite encouraging, almost always falling between two and six minutes per production. For example, at these rates, one would expect a programmer to be able to complete a 500-production expert system prototype in three to eight days, assuming six hours per day spent working on the system. This includes the time spent understanding the task description and designing, coding, testing and debugging the system. It does not include the time spent collecting domain knowledge and writing the domain-oriented task description.

The data for subject S1 in Figure 5-2 also suggests that programmers can learn TAQL very rapidly. S1 was a first-year graduate student when the experiments began. Prior to the experiments, he had written one small Soar program and translated it into TAQL. Between rounds 1 and 2, he wrote two TAQL programs, both roughly the same size as the experiment tasks. The first seven of his eight experiment tasks show a steady and rapid improvement in his encoding rate, from 12.2 minutes per Soar production to 1.6 minutes per Soar production. By his fifth task, his encoding rate is in the same range as S3's, the primary TAQL developer. S1's median encoding rate fell from 7.4 minutes per Soar production for his first four tasks to 2.9 minutes per Soar production for his last four. The 2.9 minutes per Soar production is better than S3's median rate of 3.2 minutes per Soar production for the same four tasks.

Figure 5-3 depicts the relation between task size (measured as the number of words in the task description) and encoding rate, by round. One might expect the encoding rate to slow down as the task size increases, but no such trend is apparent. This is true despite a nearly three-hundred-fold difference in size between the smallest and largest descriptions. This provides evidence for TAQL's scalability to large tasks.

Two effects could potentially offset slower encoding rates due to increased task size: improvements to TAQL between rounds, and learning on the part of subjects (the data indicates that the only substantial learning effects are for subject S1 in Round 1). Figure 5-3, however, shows no systematic relation between encoding rate and task size even within a round, even when subject S1's data from the first round is ignored.

This document uses domain-oriented description size as the measure of task size, and number of Soar productions as a measure of system size. These measures are useful because they allow one to measure the task size before it is implemented, and because productions are a common measure of system size for expert systems. However, other measures are possible, and investigating whether other measures give qualitatively different results is worthwhile.

To explore this, I considered four other measures of task size: the number of TCs in the implementation, the number of SPs the TCs compiled to, the number of symbols in the TAQL source code, and the number of symbols in the productions compiled from the source code. These units also yield three new measures of encoding rate: minutes per TC, seconds per TC symbol, and seconds per Soar-production symbol. I do not present the alternative figures here, but none of these alternative measures yielded qualitatively different results. The number of TCs proved to be somewhat less stable than the other two measures. This was expected. In TAQL, it is usually possible to either group several

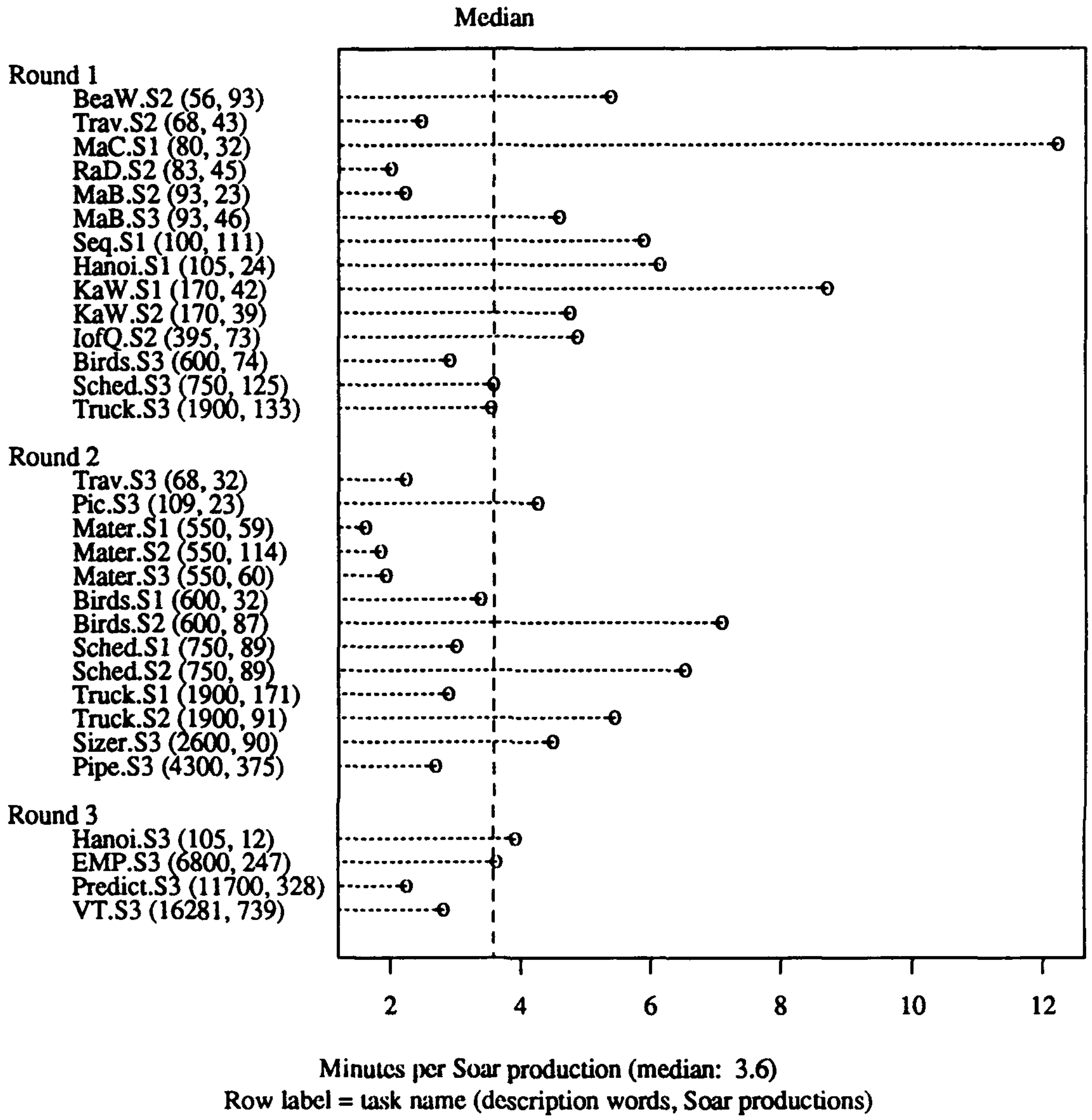


Figure 5-3: Encoding rate versus task description size (measured in words), by round

related pieces of knowledge in a single TC, or to write a separate TC for each of the pieces of knowledge. Either way compiles to the same number of productions (usually one production per piece of knowledge), but the numbers of TCs can be very different. Section 4.3.1 presents additional data on the relative quality of different measures of system size — in particular, see Figure 4-3.

Subject S3 was the primary TAQL developer and was the most experienced TAQL and Soar user of the three subjects. He also wrote most of the task descriptions (Section 4.1 lists the author of each description), though he did not look at them for some time before he used them in the experiments.[13] One might worry, then, that subject S3 had an unfair advantage and that his development times are artificially low because of it. The data does not support this suspicion. Figure 5-4 shows the total development times for the four tasks that all three subjects developed (four of the small expert system tasks). Subject S3 had the lowest development time for only one of those tasks (the trucking task), and then only by an insignificant margin. Subject S1, the most novice TAQL and Soar user, had the lowest development time for the other three tasks. Figure 5-2 shows that S3's performance on these three tasks is representative of his performance on all of the experiment tasks.

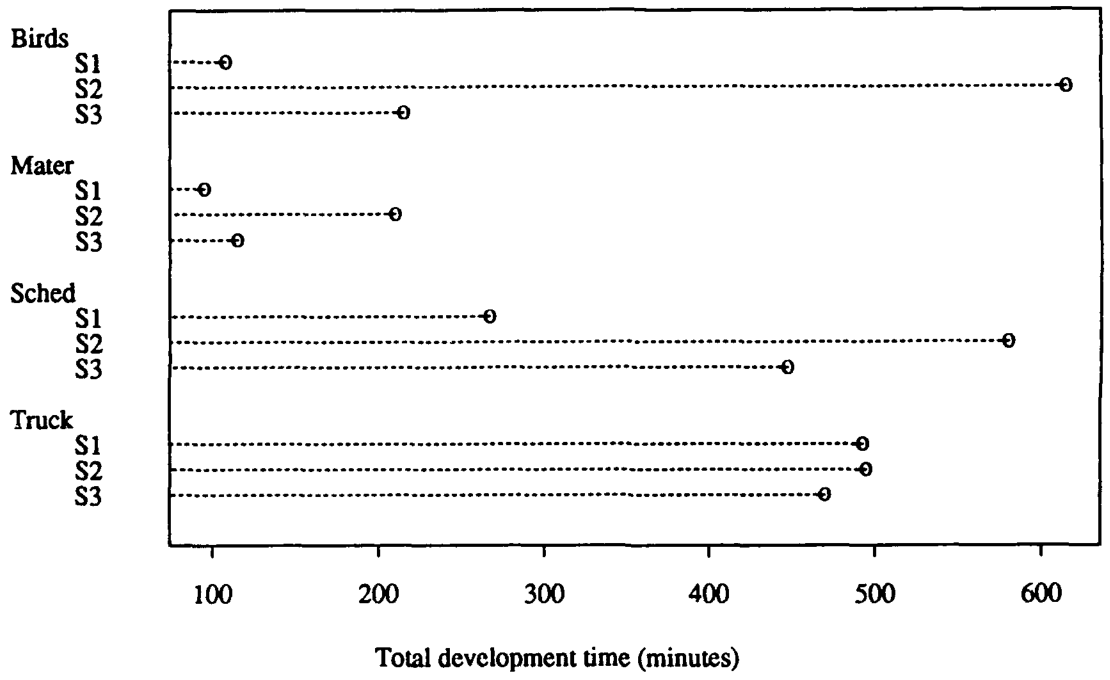


Figure 5-4: Total development times for the tasks all three subjects developed

[13]Subject S3 did not look at the descriptions he wrote for the three large expert system tasks in Round 3 for at least one and a half years before using them in the experiments.

5.3. The effectiveness of the model tools

Section 4.3.2 described a class of errors I refer to as *model errors*. These errors involve incorrectly encoding some aspect of an abstract conceptual model of a problem space, its data objects, or its communication with other spaces. Model errors accounted for nearly a third of all errors subjects detected in the TAQL experiments. Section 4.4.3.2 described the tools added to TAQL to help programmers detect and fix such errors.

This section analyzes the effectiveness of those tools. There were 89 model errors in all. This section analyzes the 76 space and data model errors for which I could determine from the logs whether or not they could have been caught by the model tools. First, the errors were separated into those that occurred before and after the model tools were developed, respectively. These two groups were further divided into the errors that could have been caught by the model tools, and those that could not have been. For each of the resulting four groups of errors, I computed the group size and average fix time.

Figure 5-5 summarizes the results. Of the 76 errors, 54 (71%) could have been caught by the tools developed for use in Round 3. Prior to building the tools, the average fix time for errors that would have been catchable was nearly identical to the average fix time for errors that would not have been catchable. After building the tools, uncatchable errors took three times longer to fix than catchable errors. I conclude that the model tools are highly effective for catching space and data model errors and that this translated into reduced debugging time.

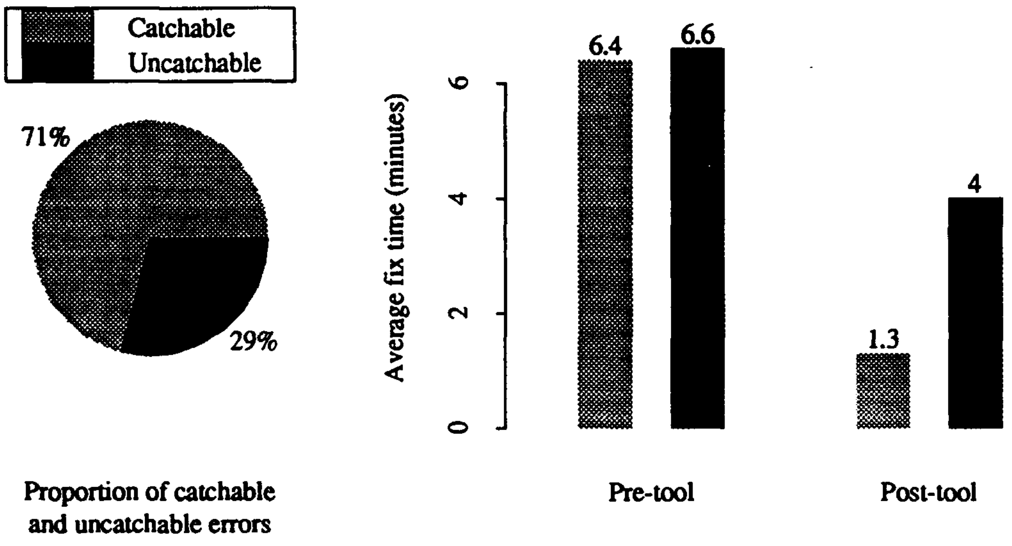


Figure 5-5: Space and data model errors before and after model tools

Chapter 6
Discussion and Conclusion

My thesis is that the same strategies that have been used to design method-based tools for narrow, strong methods can be applied profitably to much broader, weaker methods. I have supported this thesis by applying these strategies to PSCM, the problem-solving method that underlies Soar. The result is TAQL, a general-purpose expert-system development tool. TAQL has vastly broader scope than current method-based tools and yet proved to be more effective than those tools in the experiments. The following data, presented fully in Chapter 5, supports these claims:

- TAQL has been used to build dozens of systems, including analytic and synthetic expert systems and psychological models. The systems range from small prototypes to large and mature ongoing projects. TAQL users are from fields as diverse as computer science, psychology, and the social sciences.
- TAQL was used to develop two tasks that were previously built using two well-known method-based tools, KNACK and SALT. TAQL outperformed SALT by a factor of 1.3 and outperformed KNACK by a factor of 5.4.
- The encoding rate of the subjects in the TAQL experiments did not slow down as the tasks became larger, despite two orders of magnitude difference in size between the smallest and largest tasks. TAQL appears to be equally effective for small and large systems.
- The novice subject in the experiments, S1, displayed a rapid learning curve. His median encoding rate fell by a factor of 2.6 from his first four tasks to his last four. By his fifth task, he was performing at the same level as the primary TAQL developer (S3).
- The overall median encoding rate for the experiments was 3.6 minutes per Soar production. At this rate, a 500 production expert-system prototype could be completed within one work week.

These results bear out the thesis that traditional method-based-tool design strategies can yield highly-effective tools even when applied to weak problem-solving methods. Furthermore, Section 2.5.1 described research showing that strong methods of the sort used in method-based tools can readily be embedded within PSCM, yielding more of the advantages of method-based tools. This work has not yet been incorporated into TAQL, but doing so should not be difficult and should yield further performance gains.

6.1. What makes TAQL so good?

PSCM is a much weaker method than those used in existing method-based tools such as SALT and KNACK. Therefore, at the outset, I expected TAQL to approach (but not equal) the effectiveness of method-based tools on the narrow range they can encode, and greatly surpass their scope. Instead, the experiments show that TAQL both has very general scope and can outperform method-based tools even in domains they are well-suited to. It is fruitful to speculate on why this is so. I attribute TAQL's effectiveness to a combination of PSCM's simplicity and flexibility and the principled way in which TAQL was designed. I must emphasize that the material in this section is speculative, and that further research is required to verify the sources of TAQL's power.

6.1.1. Simplicity

PSCM is very simple and uniform. It comprises only a few kinds of entities: tasks, problem spaces, states, and operators. Each kind of object has a small number of well-defined knowledge roles. Finally, the possible relationships among problem spaces correspond to these knowledge roles, since the only service one problem space can perform for another is to supply knowledge that is missing from one of its roles.

There are several advantages to a simple method:

- A simple method can be clearly explicated. The tool-design strategy used for TAQL relies on having a clear understanding of the basic problem-solving method.
- A knowledge engineer has less options to consider when designing a system: everything must be represented in terms of problem spaces. In contrast, tools such as Knowledge Craft (Pepper & Kahn, 1986) and KEE (Filman, 1988b) provide a bewildering array of representations such as rules, frames, and procedural code. PSCM's simplicity does not constrain its expressiveness, however. As discussed in Section 5.1, TAQL has been used to implement a very wide variety of tasks.
- Portions of the mapping from natural descriptions of a domain to PSCM's knowledge roles can be identified, as described in Section 2.4. This makes it possible to build tools to facilitate this mapping. For example, TAQL's data types (Section 4.4.2.3) facilitate the representation portion of the mapping, and the operator-control TC (Section 4.4.2.2) facilitates the communication portion of the mapping. I surely have not completely identified the mapping functions for PSCM. Nonetheless, every part that is identified creates opportunities to build new tools that will make the mapping easier for a knowledge engineer to apply.

6.1.2. Flexibility

Traditional method-based tools force the domain expert to express knowledge in a predefined domain language. The domain language is designed to make close contact with the way domain experts actually conceive of the task. Unfortunately, the contact is never perfect and domain experts are forced to reconceive their experience in unfamiliar terms to some extent. Also, it can be impossible to use such a tool to express even slight variants of the task. Even if a tool was well-suited to a task when the system was first built, it might be difficult to modify the system to reflect future changes in the domain's methods. (Johnston, et al., 1990) discusses this brittleness problem in the context of the generic-tasks approach to method-based tools. Section 2.5.1 briefly describes their effort to overcome this problem by embedding generic tasks in Soar.

The brittleness problem is sometimes avoided by hand-coding parts of the system that the tool itself cannot build. This is likely to be extremely difficult for anyone but the tool developers, since it requires intimate knowledge of the tool's internal representations. It is akin to editing the assembly code generated by a C compiler. While it is possible for sophisticated users, it is likely to be easier to write the entire program by hand. In (Klinker et al., 1989), the designers of the KNACK knowledge acquisition tool describe their efforts to build a variety of systems using KNACK. For the eight systems they discuss, between 20 and 70 percent of the total development time went into manual refinements to the system. In one of the systems, over 80 percent of the rules were either hand-modified or hand-coded.

TAQL does not suffer from these problems. When writing the domain-oriented task description, the domain expert is free to express the task in whatever way seems natural. Also, PSCM is not a fixed, narrow problem-solving method. A knowledge engineer can design whatever problem spaces are necessary and can readily modify existing spaces to reflect changes in the domain.

6.1.3. Principled tool design

AI researchers recognized early-on that low-level programming tools are insufficient for building useful, maintainable expert systems (Clancey, 1983; Chandrasekaran, 1986; McDermott, 1986; Gruber & Cohen, 1988; McDermott, 1988; Musen et al., 1988). Effective tools should help the programmer structure the knowledge at a higher level. To achieve this, the tool designer must consider not only the tool's computational mechanisms, but how they relate to the knowledge they will be used to encode.

The approaches used to build method-based tools exemplify this philosophy. The TAQL effort has shown that the same approaches can apply profitably even to a weak problem-solving method. In building TAQL, I have: explicated a problem-solving method (PSCM) and a suitable computer representation of that method (TAQL); selected an external form in which to represent domain knowledge — domain-oriented natural-language task descriptions; and explored the mapping from domain knowledge to PSCM and ultimately to TAQL.

Finally, I have designed TAQL to closely reflect the structure of PSCM. TAQL constructs correspond to PSCM knowledge roles. The keywords within a construct correspond to the aspects of a knowledge role that must be defined for it to be meaningful. Thus, once a knowledge engineer has designed a set of problem spaces in PSCM's abstract terms, it is relatively straightforward to express that design in TAQL. The TAQL compiler converts the TAQL constructs to the Soar productions that form the executable system. Prior to TAQL, programmers had to hand-code Soar productions representing their problem-space designs. Soar productions have a very uniform structure with no clear syntactic correspondence to PSCM's structure. Therefore writing Soar productions required more expertise and discipline on the part of the programmer.

The evolution of the R1/XCON expert system illustrates both the necessity of principled tools and the usefulness of problem spaces as a knowledge-structuring formalism (Soloway et al., 1987; Bachant, 1988; Barker et al., 1989). The R1 computer-configuration expert system (McDermott, 1982) was originally written in OPS5 (R1 was later renamed XCON). Since then, it has been in continuous use at Digital Equipment Corporation. By 1988, XCON had grown from about 700 rules to over 10,000 rules. Years before that, maintaining and extending XCON had become a serious problem. OPS5 does not provide abstractions above the production level, and consequently XCON's control structure was implicit in individual rule conditions and OPS5's conflict-resolution strategies. Each programmer had developed different programming tricks for implementing higher-level control — it was often possible to tell who had written a rule just by reading it. It became increasingly difficult to modify the system and to predict the consequences seemingly-innocuous changes.

The RIME programming methodology (Bachant, 1988) was developed to remedy the problem. RIME evolved from an effort to recode part of R1 in Soar (Rosenbloom, et al., 1985). Problem spaces proved to be a structure in which the task decomposition and the functional roles of knowledge were clear. The RIME programming methodology is essentially a set of OPS5 programming conventions that layer simplified problem-space concepts on top of OPS5. Rule-checking programs ensure that programmers follow the RIME conventions. The articles cited in the previous paragraph claim that recoding XCON using the RIME methodology has dramatically improved XCON's maintainability.

The TAQL experiments showed that problem spaces are effective for initial prototype development. The RIME effort shows that the benefits of problem spaces extend throughout the software life-cycle, even for very large expert systems.

6.2. Other contributions

The primary result of my research is that the strategies that have traditionally been used to design method-based tools can be applied profitably to a much weaker method. This section lists the major additional contributions of the TAQL effort.

Soar's computational model admits effective knowledge acquisition for a wide range of expert system tasks. This is closely related to the primary result, but puts the focus on Soar. The benefit of applying method-based-tool design strategies to weaker methods depends in part on the nature of the method they are applied to. Soar's method, PSCM, has features that make the effort particularly profitable. Section 6.1 discusses these features.

The methodology used to evaluate TAQL enables high-quality quantitative comparisons among knowledge acquisition tools. The domain-oriented task descriptions used in the TAQL experiments are written solely in domain terms and are not biased in favor of any particular computational formalism. Therefore, researchers can use the same descriptions to run TAQL-style experiments on a variety of tools. Furthermore, the methodology was designed so that experiments are easy to run and to participate in, require no special equipment, and do not need to be supervised by an experimenter. I intend to publish the task descriptions, but have not yet decided on the best form. In the meantime, interested parties can order the descriptions from the Soar project as described in Appendix A. I also hope that other researchers will write domain-oriented descriptions to add to this benchmark set.

The experimental methodology provides useful guidance in tool development. I made extensive use of the data from the experiments in designing improvements to TAQL. In fact, the design philosophy was to adopt the 80-20 rule: that 80% of the benefits are often obtained from only 20% of the effort (and, conversely, that the remaining 20% of benefit requires 80% of the work). Thus, I built tools that could be used early-on, even if the initial versions were crude, and evolved those tools as dictated by their use in the experiments. The error data from the experiments was by far the most useful in discovering profitable ways to evolve TAQL.

6.3. Future work

The tool-evaluation methodology developed for this dissertation seems to be very useful and easy to apply. However, its benefits will not be fully realized until it has been used to evaluate a wide variety of tools. This is not something that can be done alone. Researchers throughout the field must come to accept that principled evaluation is absolutely necessary, and must expend the effort to perform the evaluations. I intend to actively pursue this, and hope to get other researchers to use this methodology to evaluate their own tools.

The set of task descriptions used in the TAQL experiments is a good starting point, but a larger and more diverse set written by a variety of authors is needed. I intend to encourage researchers and application developers to produce domain-oriented descriptions for inclusion in the benchmark set.

TAQL should evolve in a variety of ways. Currently, TAQL programs are traced at the Soar level, in terms of working memory changes and production firings. Tracing should occur at the problem space level, so that the user only sees a system in terms of its PSCM structure. This might seem easy to do, but it is complicated by Soar's learning mechanism (chunking). Chunking is currently defined at the Soar level rather than at the PSCM level. Therefore, learned productions (chunks) do not necessarily have an interpretation in terms of PSCM. Research is underway to understand chunking in terms of PSCM.

More tools should be built to assist the knowledge engineer in mapping knowledge from the domain to PSCM. Section 2.4 described some of the functions involved in this mapping, and some TAQL features assist with some of those functions. For example, the abstract data types (Section 4.4.2.3) assist with the representation function, and the operator-control TC (Section 4.4.2.2) assists with the communication function. But more thought needs to go into designing mapping tools. Tools that assist with the initial segmentation of a task description into problem-space components would seem particularly useful.

Currently, TAQL programmers cannot define new TAQL constructs or abstract data types. The keyword notation used in TCs has proved to be very useful in defining the information that must be supplied to instantiate a PSCM knowledge role. The same notation should be equally useful for defining task-specific templates that expand into a set of TCs. TAQL should provide a relatively simple way to define new templates and the mapping from template to TCs. Programmers should also be able to define new abstract data types, by defining a type-specific notation and defining how that notation maps to Soar's attributes and values.

Section 2.5.1 described work at Ohio State on embedding one approach to method-based knowledge acquisition in Soar. That effort showed that the embedding is not difficult and that the resulting system shares the advantages of method-based tools and general-purpose architectures. The embedding was done at the problem space level, and so incorporating this work into TAQL should not be difficult.

When PSCM has insufficient knowledge to proceed in a problem space, it automatically sets up a subspace whose task is to acquire sufficient knowledge to proceed (see Section 2.1). This suggests a way to add an interactive component to TAQL. Whenever PSCM recognizes that it has insufficient knowledge, it could set up a subspace to acquire the missing knowledge from the programmer. I call this approach *impasse-driven acquisition*. The advantage of this approach is that new knowledge is acquired in the context that it is needed. TAQL could make use of knowledge about the type of impasse and the pre-impasse situation to describe to the programmer what kind of knowledge is missing. A simple form of this should be easy to implement. For example, one can determine from the kind of impasse what kind of TC the programmer should supply. If PSCM has selected an operator but lacks sufficient knowledge to apply it, then it probably needs an apply-operator TC for that operator. Novice TAQL programmers have reported that even this limited guidance in interpreting impasses would be useful. Ideally, TAQL would interpret the impasses in terms of the domain knowledge that is missing rather than in terms of the PSCM components that are missing. Providing that level of impasse interpretation is an open research problem.

Section 4.4.4 lists additional potential TAQL improvements that were suggested by the results of the final round of TAQL experiments.

The current TAQL tutorial materials introduce basic problem-space concepts. After completing the tutorial, programmers should be able to build simple multiple-space TAQL systems. The materials should be extended to cover more advanced topics such as learning, using data types and the the operator-control TC, and using the space and data model facilities.

6.3.1. A new expert-system development paradigm

The use of domain-oriented task descriptions in the TAQL experiments suggests a new expert-system development paradigm. In this paradigm (depicted in Figure 6-1), the domain expert writes a domain-oriented task description, possibly with the assistance of a professional writer. The knowledge engineer maps this knowledge to PSCM, creating a design for a set of problem spaces implementing the task. Finally, the knowledge engineer maps the design to TAQL code and tests it.

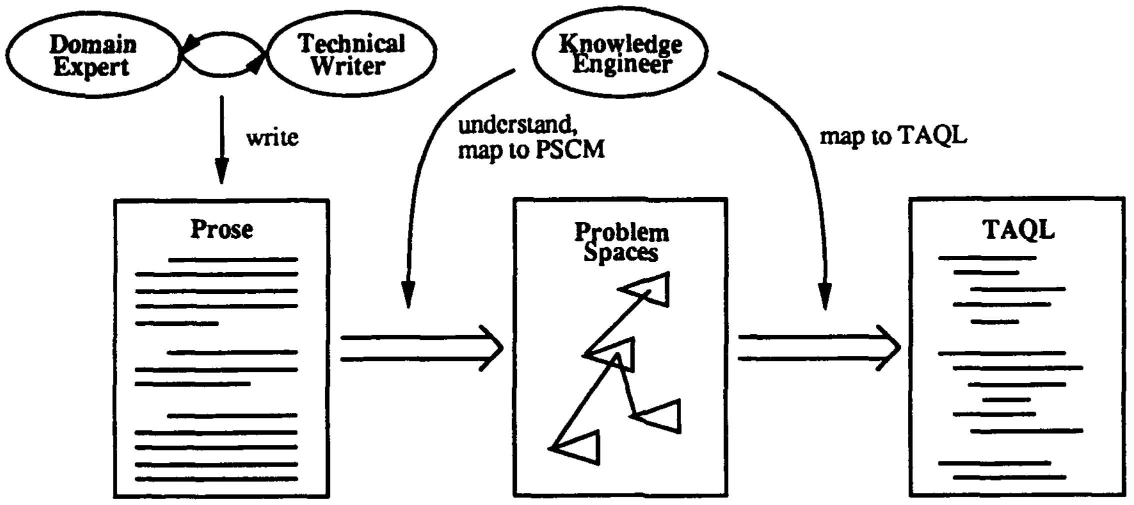


Figure 6-1: The TAQL expert-system development paradigm

The initial domain-oriented description is likely to be incomplete, unclear, or incorrect. Thus, later parts of the design, coding and testing will require cooperation between the domain expert and the knowledge engineer to complete the description.

I *do not* have any data that would let me compare this development paradigm to existing paradigms. However, I see enough potential benefits to warrant further exploration. The potential benefits include:

- The knowledge engineer may not require as much assistance from the domain expert in the early development stages. The domain-oriented description will provide much of the knowledge, even if it is incomplete.
- Since the task description is informal prose, it should be easier to write than a formal program. Therefore writing the description is a useful intermediate organizing step.
- A well-organized description could make it easier to grasp the overall structure of a task, and so makes design and coding easier. It could also serves as task-level documentation once the system is built. This could ease maintenance.
- The descriptions can be written by a technical writer working with the domain expert, freeing programmers from much of the burden of knowledge elicitation so that they can focus on what they do best. This is particularly important considering the high cost of programmer time.
- Technical writers specialize in organizing and presenting information clearly. Adding a technical writer to the development effort creates a team of cooperating specialists, instead of requiring a jack-of-all-trades knowledge engineer who is likely to be highly-skilled in only some of the required trades.

- The domain expert can do considerable knowledge-level debugging while writing the task description, before the knowledge engineer ever sees it. In writing the domain-oriented descriptions for the TAQL experiments, I found that many knowledge-level errors are easy to detect in the descriptions — errors that could be very difficult to detect in code. For example, missing table entries stand out in a task description but are often easy to overlook in their computer representations.

This expert-system development paradigm has elements in common with the KADS approach to knowledge engineering (Hayward et al., 1988; Wielinga et al., 1991). A fundamental premise of the KADS work is that expert-system development should be driven by the demands and structure of the task, rather than by the demands and structure of any particular implementation media or computational model. In KADS, expert-system development is viewed as a modeling activity. Starting at the knowledge level, knowledge is successively categorized and refined into a set of models reflecting different aspects of the task at different levels of abstraction. These include models of the task, the application, the expertise, and the system design. Transformations from knowledge-level models to symbol-level models are *structure-preserving*. That is, they involve *adding* symbol-level information to a conceptual model rather than performing a wholesale reformulation of the knowledge. Thus, the structure of the implementation reflects the structure present at the knowledge level.

The paradigm proposed here shares the fundamental KADS premise stated in the last paragraph. The domain-oriented task description is written in natural language, in whatever way the domain expert feels is clear and convenient, without regard to (or even knowledge of) the final implementation media (PSCM and TAQL). The TAQL paradigm also has several levels of models. The task description serves as a knowledge-level model. The abstract PSCM formulation of a task is a mixed knowledge-level and symbol-level model. The TAQL code is a pure symbol-level model. Furthermore, as discussed in Section 2.4, the transformations from task description to PSCM to TAQL are structure-preserving in the KADS sense. PSCM components are identified by a functional segmentation of the task description. Therefore, the structure of the problem spaces mimics the structure of the task as presented in the description. Knowledge-level terms remaining in the PSCM model are replaced by appropriate symbol-level representations in the TAQL code. This corresponds to the KADS notion of simply adding symbol-level structure rather than reformulating knowledge wholesale.

6.3.2. A day in the life of Louie

We are at the end of this final section, a section of hopes and speculations. It seems fitting to end with a tiny glimpse into a possible future — into a day in the life of Louie Stephens, knowledge engineer.

May 15, 1997: Pittsburgh, PA. *Louie Stephens squinted groggily at his alarm clock. 7:00 am. Time to get up. As he shuffled towards the coffee maker, he reflected on the task that he would face today and for the next several weeks. The deadline for entries in the 1997 Knowledge Acquisition Review was coming up on July 18.* Soarware Engineering, Inc. *had selected him as one of the TAQL programmers that would participate in this year's task-development experiments. Over the next month, he and the other programmers would each be assigned several tasks randomly drawn from a standard pool of domain-oriented task descriptions. Researchers all over the world would be doing the same with their tools. A few years ago this was unheard of. Now, it was difficult to have your ideas taken seriously if you could not back them up with data published in the Review.*

To be sure, the experimental procedures were still in their infancy. Some of the task descriptions had not yet been validated, and the pool of benchmark tasks was not yet as diverse as it should be. But this was changing. More and more researchers that were dissatisfied with the benchmark set were submitting new descriptions. In time, it would settle down. In the early years of the Review, noisy data was another problem — too many external factors influence software development to realistically control for them all. But that too was changing. As the pool of data grew, these external influences began to average out, and the data began to show strong and enlightening regularities. Already these regularities were strong enough to let tool builders identify some of the most promising approaches.

8:00 am. *Louie arrived at Soarware Engineering's world headquarters near the CMU campus and sat down at his workstation. Allen had left the description for the experiment task he was to begin today on his desk. He created a file for his experiment log, and after jotting down the time and a brief notation he began to read the description. Elevator configuration, one of the mid-sized tasks in the benchmark pool.*

An hour later he had finished his first read of the description, and went to get a cup of coffee after making another note in his log. Of course, he didn't entirely understand the task, but he thought he had a good feel for what it involved, and knew where to look for the details when he needed them. On returning, he started Soar on his workstation. When he began working for Soarware Engineering three years ago, Soar and TAQL were loosely-coupled. Now, the two had merged and had a single interface. Programmers could still write Soar productions or TAQL constructs, but programming was now largely visual. Louie selected the problem-space editor from a pull-down menu.

He looked at the notes he had taken while reading the description, and created a labeled problem-space icon corresponding to each major activity involved in elevator configuration. It was already clear how most of the spaces fit together, and he connected these spaces with labeled lines. The others he would have to think about harder. There

- The domain expert can do considerable knowledge-level debugging while writing the task description, before the knowledge engineer ever sees it. In writing the domain-oriented descriptions for the TAQL experiments, I found that many knowledge-level errors are easy to detect in the descriptions — errors that could be very difficult to detect in code. For example, missing table entries stand out in a task description but are often easy to overlook in their computer representations.

This expert-system development paradigm has elements in common with the KADS approach to knowledge engineering (Hayward et al., 1988; Wielinga et al., 1991). A fundamental premise of the KADS work is that expert-system development should be driven by the demands and structure of the task, rather than by the demands and structure of any particular implementation media or computational model. In KADS, expert-system development is viewed as a modeling activity. Starting at the knowledge level, knowledge is successively categorized and refined into a set of models reflecting different aspects of the task at different levels of abstraction. These include models of the task, the application, the expertise, and the system design. Transformations from knowledge-level models to symbol-level models are *structure-preserving*. That is, they involve *adding* symbol-level information to a conceptual model rather than performing a wholesale reformulation of the knowledge. Thus, the structure of the implementation reflects the structure present at the knowledge level.

The paradigm proposed here shares the fundamental KADS premise stated in the last paragraph. The domain-oriented task description is written in natural language, in whatever way the domain expert feels is clear and convenient, without regard to (or even knowledge of) the final implementation media (PSCM and TAQL). The TAQL paradigm also has several levels of models. The task description serves as a knowledge-level model. The abstract PSCM formulation of a task is a mixed knowledge-level and symbol-level model. The TAQL code is a pure symbol-level model. Furthermore, as discussed in Section 2.4, the transformations from task description to PSCM to TAQL are structure-preserving in the KADS sense. PSCM components are identified by a functional segmentation of the task description. Therefore, the structure of the problem spaces mimics the structure of the task as presented in the description. Knowledge-level terms remaining in the PSCM model are replaced by appropriate symbol-level representations in the TAQL code. This corresponds to the KADS notion of simply adding symbol-level structure rather than reformulating knowledge wholesale.

6.3.2. A day in the life of Louie

We are at the end of this final section, a section of hopes and speculations. It seems fitting to end with a tiny glimpse into a possible future — into a day in the life of Louie Stephens, knowledge engineer.

were only a few he hadn't placed yet, and all of them were related to fixing constraint violations. After about half an hour of scratching out possible designs on paper, he thought he had a workable way to fix constraint violations. He modified his earlier problem-space diagrams to reflect the more detailed design. Now all the pieces fit together. Now it was time to work out the spaces in more detail.

Louie clicked on the top-level problem space. It seemed to be a good place to start. A new window appeared displaying the contents of that space. The window had subdivisions for each of the PSCM knowledge roles that he had to define. The subdivisions were mostly empty, but there were already two operators listed there. TAQL had inserted them automatically, because two of the spaces that were attached to the top space in his diagram were tagged as operator-application spaces. After naming the two operators, he created two more operators that he knew he would need but that would probably be simple enough that he would not need separate problem spaces to apply them in.

Before filling in the rest of the spaces, he needed to design the data representations he would use. He called up the object library editor. Initially the library just contained the predefined object types: goals, evaluations, lists, text, and so forth. He created several more object types and defined the attributes of each. He also defined the state attributes for the top space.

Next he clicked on the search control subdivision and opened an operator-sequencing editor. The editor let him draw control diagrams for the operators and attach conditions to the links. The editor made it very easy to see how a space's operators fit together. It was once common to forget to encode parts of the control knowledge, but bugs of that sort were now rare. The tool even provided a library of predefined control diagram corresponding to common methods. Programmers could use these methods as-is or edit them to create their own variations. At one point Louie mistyped an attribute name on one of the conditional transitions between operators. He immediately got a warning that no such attribute was defined for the type of object he had attempted to use it with.

After lunch he filled in the remaining knowledge subdivisions in the space: operator proposals and applications, goal tests, the initial state, and so forth. When he was done, he clicked the button that invoked the problem space checker. It examined the information he had entered to try to find missing or incorrect items. It couldn't catch every error, naturally, but it was rather sophisticated. It informed him that one of his operators created an attribute that wasn't used anywhere else in the space. He realized that one of the other operators should have used that information, but that he had left out part of the operator application for that operator. Louie added the missing information, ran the checks again, and got no warnings.

Louie continued defining problem spaces for the rest of the day. The interface caught many errors right as he made them — errors that would have been difficult to detect, diagnose and fix at run time, when the code was no longer fresh in his mind. By the end of the day, he had the basic structure of all of the problem spaces coded. Tomorrow he would fill in the remaining domain knowledge.

Stretching, he rose to leave the office. Not a bad day, he thought. Not bad at all.

Appendix A
Obtaining TAQL, Soar, and Related Documents

The following information is excerpted from the manual entry for Soar:

Soar is an architecture for a system that is to be capable of general intelligence. Soar is to be able to: (1) work on the full range of tasks, from highly routine to extremely difficult, open-ended problems; (2) employ the full range of problem-solving methods and representations required for these tasks; and (3) learn about all aspects of the tasks and its performance on them. Soar has existed since mid-1982 as an experimental software system (in OPS5 and Lisp), initially as Soar 1, then as Soar 2, and currently as Soar 5.2.2. Soar realizes the capabilities of a general intelligence only in part, with significant aspects still missing.

Soar, a memory intensive Common Lisp program, has been successfully ported to a variety of Common Lisp implementations: Lucid (Mach/VAX, SunOS), Xerox Common Lisp, TI explorer Lisp, Symbolics Common Lisp, KCL, Andrew Lucid, Allegro on the MAC II, Suns and DECstations, and CMU RT Lisp.

The Soar software (5.2.2) is in the public domain. This software is made available AS IS, and Carnegie Mellon and The University of Michigan, make no warranty about the software, its performance, or the accuracy of our documentation in describing the software. All aspects of Soar are subject to change in future releases.

The Soar group is widely distributed and communicates largely by electronic mail. We maintain an open mailing list, soar-announcements@cs.cmu.edu, on which we distribute general information about Soar and external releases.

General requests for information about Soar may be sent to soar-requests@cs.cmu.edu, requests for documentation may be addressed to soar-doc@soar.cs.cmu.edu. Physical mail requests should be sent to:

The Soar Group
Carnegie Mellon University School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA 15213-3890
USA

TAQL is also in the public domain as of release 3.1.4. It is included as part of the standard Soar distribution. The distribution also contains on-line versions of the TAQL tutorial materials, including example programs. Most TAQL and Soar related papers listed in the *References* section can be obtained from the Soar group as described above.

The task descriptions used in the TAQL experiments can also be obtained from the Soar group.

Appendix B

TAQL Code for the Light-Toggling Example

This appendix gives the complete commented TAQL code for the light-toggling example presented in Section 2.2.1.

```
;;;; Gregg Yost
;;;; March 12, 1992

;;; A very very simple example TAQL program.  Turn a light on if it is
;;; off; turn it off if it is on.

;;;;===========
;;;; Task setup
;;;;===========

;;; ---- propose task operator:

(propose-task-operator top*pto*light-task
  :op light-task)

;;; ---- propose task state:

;; The top state initially has a signal-light that is turned off.
;;
(propose-task-state top*pts
  :new (signal-light ((light ^status off))))

;;;;===========
;;;; Task space
;;;;===========

;;;------------------------------
;;; Problem space setup and return
;;;------------------------------

;;; ---- propose space:

;; The light space performs the light-toggling task, by applying
;; the light-task operator.
;;
(propose-space light*ps
  :function (apply operator light-task)
  :space light)
```

```
;;; ---- propose initial state:

;; Create a duplicate of the top state's signal light on the initial
;; state.
;;
(propose-initial-state light*pis
  :space light
  :copy (:copy-new (signal-light) :from superstate))

;;; ---- goal test:

;; The task is done when the light on the state has a status different
;; from the light on the top state.
;;
(goal-test-group light*gtg*status-changed
  :space light
  :group-type success

  :when ((state ^signal-light <local-light>)
         (light <local-light> ^status <status>)
         (superstate ^signal-light <top-light>)
         (light <top-light> ^status <> <status>)))

;;; ---- return result:

;; When we're done, copy the signal light from the subspace to the
;; superspace, replacing the old light on the superstate.
;;
(result-superstate light*rs
  :space light
  :group-type success

  (edit :what superstate
        :copy (signal-light :remove target)))

;;;;-----------------
;;;; Propose operators
;;;;-----------------

;; Propose toggling a light whenever there is one on the state.
;;
(propose-operator light*po*toggle-light
  :space light
  :when ((state ^signal-light <light>))
  :op (toggle-light ^light <light>))
```

```
;;;;----------------
;;;; Apply operators
;;;;----------------

;; Toggle a light by replacing its status by on or off, as appropriate.
;;
(apply-operator light*ao*toggle-light
  :space light
  :op (toggle-light ^light <light>)

  (edit :what (:none light <light>)
        :replace (status :by off
                         :when ((light <light> ^status on)))
        :replace (status :by on
                         :when ((light <light> ^status off)))))
```

Appendix C
A Sample Domain-oriented Task Description

This appendix presents a sample domain-oriented task description. The description is the one used for the shipment scheduling assistant task in the TAQL experiments (see Section 4.1). This description was derived from a formal description of a larger set of problems in the shipment scheduling domain (Filman, 1988a). The formal description is for the problem used as an example throughout a recent KEE paper (Filman, 1988b).

C.1. A shipment scheduling assistant

The Big Giant Trucking Company ships materials among cities in the Midwest. Customers contact the company and request that goods be transported from one city to another on a specified day. Big Giant's dispatchers collect the orders and create suitable delivery schedules.

A schedule consists of some number of *trips*, where each trip has an *itinerary*, a *truck*, and a *driver*. An itinerary is a list of cities and the highways that the driver should take from one city to the next. The itinerary also states what shipments, if any, the driver should pick up and deliver at each city (sometimes the driver will just pass through with no pickups or deliveries).

Producing a schedule is very difficult because of the many constraints on the trips and the schedule as a whole. For example, each driver can drive only one trip, and union drivers can only drive trips that take less than eleven hours. The shipment scheduling assistant takes the set of itineraries on the schedule (which we assume were put together by a dispatcher), and tries to find an assignment of trucks and drivers to the itineraries that does not violate any constraints. If no such assignment is possible, the assistant informs the dispatcher, who must then revise the itineraries and try again.

The remainder of this description provides the information the assistant needs to try to find valid truck and driver assignments.

C.1.1. Drivers, trucks, cities, and highways

Tables C-1, C-2, and C-3 define the driver, trucks, and highways used by Big Giant.

Big Giant serves the following cities:

- *In Illinois:* La Harpe, Oregon, Thayer, Utica, Viola, Yale, and Zion.
- *In Indiana:* Attica, Bloomington, Cook, Delphi, English, Fowler, Gary, Hebron, Indianapolis, Jasper, Kokomo, Mitchell, New Harmony, Paoli, Roselawn, Seymour, and Warsaw.

Name	**Union**	**License Class**
Brown	yes	3
Gray	no	1
Green	yes	3
White	no	2

Table C-1: Drivers

Name	**Class**
Cannonball	big
Piper	small
Queen Bee	medium
Traveler	medium

Table C-2: Trucks

C.1.2. Constraints

The constraints on schedules and trips are:

- Each driver can drive only one trip, and each trip has only one driver.
- Each truck can be used on only one trip, and each trip has only one truck.
- The maximum weight of a truck's load at any point during a trip cannot exceed the truck's rated weight limit. Big trucks can hold 32000 pounds, medium trucks 10000 pounds, and small trucks 5000 pounds.

Name	Connects	Grade	Length
I64a	New Harmony, Jasper	primary	60.0
I64b	English, Jasper	tertiary	30.0
I65	Seymour, Indianapolis	primary	60.0
I70a	Thayer, Yale	primary	150.0
I70b	Indianapolis, Yale	primary	90.0
I74	Indianapolis, Attica	primary	60.0
I80a	Viola, Utica	primary	100.0
I80b	Cook, Utica	primary	90.0
I90	Oregon, Gary	secondary	100.0
I94	Zion, Gary	primary	60.0
S125	La Harpe, Thayer	tertiary	80.0
S25	Delphi, Attica	tertiary	40.0
S26	Delphi, Kokomo	tertiary	30.0
S37a	Bloomington, Indianapolis	secondary	50.0
S37b	Bloomington, Mitchell	tertiary	30.0
S37c	Paoli, Mitchell	tertiary	10.0
S37d	Paoli, English	tertiary	10.0
U231	Cook, Hebron	tertiary	20.0
U24	La Harpe, Fowler	secondary	180.0
U30	Gary, Warsaw	secondary	70.0
U31a	Kokomo, Warsaw	secondary	70.0
U31b	Kokomo, Indianapolis	primary	40.0
U41a	Cook, Gary	secondary	20.0
U41b	Cook, Roselawn	secondary	20.0
U41c	Fowler, Roselawn	secondary	30.0
U41d	Fowler, Attica	secondary	30.0
U41e	Yale, Attica	secondary	90.0
U41f	Yale, New Harmony	secondary	70.0
U50	Mitchell, Seymour	tertiary	30.0
U51	Utica, Oregon	secondary	40.0
U67	Viola, La Harpe	secondary	50.0

Table C-3: Highways

- The maximum volume of a truck's load at any point during a trip cannot exceed the truck's rated volume limit. Big trucks can hold 1280 cubic feet, medium trucks 640 cubic feet, and small trucks 400 cubic feet.
- The driver and truck assigned to a trip must be in the trip's origin city to begin with.
- The license class of a driver must be at least as great as the license class required by the truck he or she is assigned. Big trucks require class 3 licenses, medium trucks require at least class 2 licenses, and small trucks require at least class 1 licenses.
- A driver can only drive trips whose duration is less than his or her maximum allowable driving time. The duration of a trip is the sum of the driving times for each segment on the itinerary, plus the time needed for loading and unloading shipments during the trip. The driving time for a segment is the length of the road used for that segment divided by the estimated speed for that road (as determined by the weather and road grade, see Table C-4). Union drivers can be on a trip for at most 11 hours, while non-union drivers can be on a trip for at most 12.5 hours.
- White cannot drive on any trip that passes through a city in Illinois (he is wanted for a crime there).

Road grade → **Weather ↓**	**Primary**	**Secondary**	**Tertiary**
Fair	60	55	50
Rain	55	50	35
Snow	45	40	30

Table C-4: Estimated travel speed, given road grade and weather

C.2. Test cases

This section presents a simple test case that you can use to partially test your scheduling assistant.

The weather throughout Big Giant's area of operations is rainy. Drivers Brown and Gray are in Gary, and drivers Green and White are in Indianapolis. Trucks Piper and Traveler are in Gary, and trucks Cannonball and Queen Bee are in Indianapolis.

The dispatcher's schedule has three trips. The shipments referred to in the trips are listed in Table C-5.

- *Trip 1:* Starting in Gary, pick up the typewriter shipment and take highway U30 to Warsaw, followed by U31a to Kokomo, U31b to Indianapolis, and I74 to Attica. Deliver the typewriter shipment in Attica.
- *Trip 2:* Starting in Gary, take U41a to Cook, I80b to Utica, and I80a to Viola. Pick up the carpet shipment in Viola. Then, take I80A back to Utica, and U41a to Cook. In Cook, deliver the carpet shipment and pick up the newsprint shipment.
- *Trip 3:* Starting in Indianapolis, take I70b to Yale, then take U41e to Attica.

One valid solution for this test case is to assign Gray/Piper to trip 1, Brown/Traveler to trip 2, and Green/Cannonball to trip 3. Piper is the only truck in Gary that Gray is licensed to drive, leaving Traveler (the only other truck in Gary) for Brown. White cannot drive trip 3, because it passes through Illinois, so Green must do it.

Material	Origin	Destination	Weight	Volume	Loading Time	Unloading Time
Bicycles	Roselawn	Bloomington	500.0	100.0	0.2	0.25
Books	Oregon	Mitchell	1000.0	50.0	0.2	0.25
Carpet	Viola	Cook	500.0	100.0	0.2	0.25
Computers	Seymour	Thayer	1000.0	150.0	0.2	0.25
Newsprint	Cook	Indianapolis	6000.0	400.0	0.2	0.25
Refrigerators	Kokomo	Warsaw	9000.0	600.0	0.2	0.25
Toys	La Harpe	Oregon	1000.0	100.0	0.2	0.25
Typewriters	Gary	Attica	1000.0	200.0	0.2	0.25

Table C-5: Shipments

Appendix D
A Sample Experiment Log

This appendix contains the log created by subject S3 while implementing the EMP hardness evaluator in the TAQL experiments. The portion of the log after the *Development Log* heading is the information the subject recorded during development. (Subject S3 kept his original log on paper. This is a transcription.) The remainder of the log was created after completing the implementation.

The two-letter codes preceding bug entries in the log (for example, *TY* and *ME*) are bug classification codes. These were added by the experimenter as part of analyzing the raw log. See Appendix E for definitions of these codes.

```
RTAQ Experiment Log

Date started:     3-Jun-91
Task Name:        EMP Hardness Critic and Report Generator
Builder:          S3
Soar Version:     Soar 5.2.0 (18-Feb-91 patch file)
TAQL Version:     TAQL 3.1.4 (5-29-91 internal), used structured editor
Sessions:         12
Learning:         Off

--------
The Task
--------

Evaluate an electro-mechanical design with respect to EMP hardness.
If the design is not EMP-hard, suggest appropriate design
modifications.  Generate a report on the design and its evaluation in
the standard form the government expects.  See
descriptions/dpr/kl-dpr.{mss,ps} and
descriptions/dpr/report-hard.{mss,ps} for a full description.

------------------------------------
Initial Task Acquisition Statistics:
------------------------------------

Number of development sessions:  12

Times (minutes):

  Builder task acquisition        71
  System design                  194
  Coding                         459
  Debugging and testing         172
  --------------------------------
  Total                         896

Task description size:

  6800 words.  I was familiar with the task, but had not looked at it
  since I wrote the description over two years ago.

Encoding size:

  5 problem spaces, 14 operators

  Space            Number of operators
  -----------------------------------
  EMP                    4
  Ask-question           3
  Fix-violation          3
  Gen-report             4
  Print-nested-loop      3

  Number of TCs:  233
  Number of productions:  247 (+ 198 in default, taql-support, and
                                    text library)

Encoding rates:

  These are computed based on the total time of 896 minutes (14:56 hours):

    Minutes per TC:          3.8
    Minutes per production:  3.6

------------
Print-stats:
------------

These are the taql-stats and print-stats from the run in emp.ita.trace.
Note that MIXO is actually a DECStation 5000 series, not a DEC 3100.

<cl> (taql-stats)

TAQL 3.1.4 (internal release)
Created May 29, 1991

TAQL statistics on June  10, 1991
Allegro CL 3.1.12 [DEC 3100] (3/30/90) DEC 3100  id: 385 Ultrix
MIXO.SOAR.CS.CMU.EDU

235 TCs (233 user, 2 default)
    compiled into 276 productions (247 user, 29 default)

<cl> (print-stats)

Soar 5.2.0 (internal release)
Created August 7th, 1990

Run statistics on June  10, 1991
Allegro CL 3.1.12 [DEC 3100] (3/30/90) DEC 3100  id: 385 Ultrix
MIXO.SOAR.CS.CMU.EDU

445 productions (31875 / 185640 nodes)
 0 chunks (0 / 445 productions)
```

```
2309.618 seconds elapsed
907 decision cycles (2546.4365 ms per cycle)
3629 elaboration cycles (636.4337 ms per cycle)
    (4.0011024 e cycles/d cycle)
15567 production firings (148.36629 ms per firing)
     4.2896113 productions in parallel
41696 RHS actions after initialization (55.391834 ms per action)
421 mean working memory size (3455 maximum, -95 current)
3430 mean token memory size (23926 maximum, 1625 current)
256101 total number of tokens added
254476 total number of tokens removed
510577 token changes (4.523545 ms per change)
     (12.244934 changes/action)
```

Comments on Tools:

----

1. Hard to find duplicate TC names and premature end of file. Frank's Soar mode comes with something to help the latter.

2. Should have an easier way to load libraries. One thing it should do is not try to reload if it is already loaded, as the libraries are sticky, and you get redeclaration error messages.

3. Generated variables that begin with << when variable class used. Confusing to look at.

4. Several places I had to use :directive in ways it wasn't intended to get proper sequencing of Soar actions (in write prompt/read response situations). Should be a sanctioned way to do this. The :actions keyword I'm going to add for John and Paul will do the trick.

5. It would be nice if TAQL could figure out what operator you were talking about in (superoperator ...) conditions, so that it could check the attributes. You can detect these if you think to print out the inferred data model and check operator *unknown*, but it is easy to forget to do that, and easy to not realize that one of the attributes it lists shouldn't be there.

6. Unrestrained use of the new syntax requires some degree of declarations. If you are operating in the mode of code-->print-inferred-model-->check-modify-and-install, this discourages using the new syntax initially.

7. My experience from this task leads me to believe that for sizable systems, data models should be entered manually before coding. Having to think about what your objects are like helps get things clear, and having the declarations there while coding is a useful reference. Plus, when you look through inferred data models after a long coding session, it is easy to forget what you had intended some things to look like, and thus not spot errors. Once you have most declarations in, using the inferred model is not so bad, because you can do (print-data-model (inferred) :exclude (declared)) and only see the new (or erroneous) undeclared stuff that you might have to merge into your declarations.

8. For attributes that take values from a limited set, maybe the printer should print out defenumerations for the limited sets (which users could then edit, or whatever). Right now, it is too tempting to just leave it as :type primitive, with the resulting loss of

```
    error-checking ability.

 9. There should be a better way to share declarations among segments.
    Right now the only options are to either make the declaration global,
    or repeat it in each segment it is used in.

10. It would be nice to have a way to print out an object in a form
    that could be inserted as an action.  This would make certain kinds
    of testing easier, where you want it to start out in a state
    somewhere down the line and go from there.

11. The combinatorics of a few productions in the taql-support for
    operator-control are rather severe when you have a bunch of
    operators proposed.  This scares me off from using operator-control
    in some cases.  This isn't really a TAQL problem, but a Soar reorderer
    problem -- it tests the two conditions for the two operator
    acceptable preferences first, before testing any of the conditions
    that constrain one of the operators.  These productions that are
    causing problems now would actually be quite cheap if it did
    some sort of depth-first condition ordering.

12. All in all, the models were a big win for this task.  Also, the
    text data type was a VERY big win.  It would have taken longer
    to design, and potentially much longer to code, without out it.

13. I used entailments to implement all my formulas.  That means
    I got "truth maintenance" as the user makes design modifications
    for free.

14. I had to write a few lisp functions -- one to compute logs for one
    of the formulas (which the compute function can't handle), and
    a few more to open and close files for IO.  This is something that
    could have been much more of a pain in Soar6.  Is it still going to
    allow RHS function calls?  The file IO stuff is something that could
    presumably be packaged up an put into a library.

----------------
Development Log:
----------------

I saved the scratch paper I used during the development, and have it in a
folder.  The original code and a trace are in
emp/emp.ita.{taql,trace}, and the generated report is in emp/report.{mss,ps}.

1. 3-Jun-91

   1540: begin builder task acquisition
           I was familiar with the task, but hadn't ever coded it.

   1615/ first pass read done, short break

   1650: resume builder task acquisition

   1726/ builder task acquisition done

2. 4-Jun-91

   1535: begin design
          First just the question asker and design evaluator.  Un-hardness
          fixer and result printer later.

   1637/ break
```

```
          Have data representations designed

     1741: resume design

     1843: continue design
          Have question asker and design evaluator designed.  Do a very
          rough design of the fixer, to be reasonably confident it will
          fit in.

     1935/ end design
          Ok, I'm reasonably confident I'll be able to stick with this
          design all the way through.

  3. 5-Jun-91

     1155: begin coding

     1224/ short break

     1237: resume coding

     1325/ short break

     1335: resume coding

     1501/ break
          Question proposals and tables done, formulas almost done.

     2017: resume coding

     2100/ coding done, break
          All coded except for fixer and printer.

     2200: begin test/debug
          Just want to get it to load tonight, and check model data.

  TY 2200: Bug #1:  Unlinked condition errors
     2201: Bug #1 fixed
          [1 min.]

  TY 2201: Bug #2:  Duplicate TC names
     2205: Bug #2 fixed.  This is still a hard thing to track down.
          [4 min.]

  TY 2205: Bug #3:  Premature EOF
     2207: Bug #3 fixed.  Hard to locate.  But Frank's Soar mode has a function
          for finding these, check it out later.
          It loads now, I'll check/install the data/space models tomorrow.
          [2 min.]

     2207/ break

  4. 6-Jun-91

     1437: resume test/debug
          Get space/data models checked and installed first.

  ME 1442: Bug #4:  Attribute on wrong object, caught in inferred model
     1443: Bug #4 fixed.
          [1 min.]

  ME 1445: Bug #5:  Wrong attribute name, caught in inferred model
```

```
   1446: Bug #5 fixed.
         [1 min.]

ME 1505: Bug #6:  Missing application for emp-done, caught in inferred model
   1506: Bug #6 fixed.
         [1 min.]

   1511: continue test/debug
          Models checked and installed.  Loads ok.  Begin testing.

ME 1513: Bug #7:  print-text not applying first
   1517: Bug #7 fixed.  Had wrong att name in (superoperator ^att ...),
         which the model code couldn't detect because the operator name
         wasn't there.
         [4 min.]

ME 1517: Bug #8:  Attribute tie, left out parallel pref.
   1518: Bug #8 fixed.
         [1 min.]

SE 1524: Bug #9:  Not reading replies properly?
   1532: Bug #9 fixed.  Turns out to be Soar's foolishness of compiling
         ^att (read) + & into ^att (read) + ^att (read) &.
         [8 min.]

DE 1536: Bug #10:  No way to respond 'none' to a question for a list
         of values.
   1550: Bug #10 fixed.
         [14 min.]

IE 1552: Bug #11:  Ask-question returning bad values.
   1555: Bug #11 fixed.  Had spurious apply-operator edits left over from
         an outdated piece of my design.
         [3 min.]

   1555/ break

   1720: resume test/debug

IE 1533: Bug #12:  Ask-question not properly returning values when
         ^find-id-for-value true.
   1741: Bug #12 fixed.  Had a case wrong in apply-operator, because of
         confusion in implementing my overloaded design of
         ask-question.
         [8 min.]

   1817/ continue test/debug, break
          Ok, this part works.  Between 1741 and now was a long unattended
          run with no glitches.

   1850: resume test/debug
          See if it gets the right results when it has cable shields and
          TPDs.

TE 1853: Bug #13:  Break through to lisp, there is a complex number coming
         from somewhere and RETE barfs on it.
   1920: Bug #13 fixed.  There was a sign error in one of the formulas in
         the task description.
         [27 min.]

   1920/ break
```

```
5. 7-Jun-91

    1210: begin design
           Check and refine design of fixer.

    1217: begin coding
           Design of fixer seems fine, and detailed enough to code
           from.

    1303: begin test/debug

IE  1323: Bug #14:  Not deleting fixes right.
    1324: Bug #14 fixed.  Wrong att name in goal test (but an attribute
          that was valid in this space -- this was more of a semantic
          error).
          [1 min.]

    1335/ continue test/debug
           It works!  Between 1324 and now was an uninterrupted and unattended
           run with no glitches.  I had originally decided to call initial task
           acquisition done at this point, since the actual report generation
           seemed to me so simple that it would best be done as a back end
           in some more conventional language.  But then I decided that for
           me to be able to make more solid comparisons to the KNACK DPR
           WRINGER, I had to code the report generator as well.  So,
           onward.

6. 9-Jun-91

    1210: begin design
           Design report generator.

    1221: begin coding

    1324: begin test/debug

ME  1328: Bug #15:  Unexpected operator tie in print-nested-loop
    1331: Bug #15 fixed.  Left out worst pref for done op.  I'm calling
          this a model error rather than an implementation error because
          this is something that wouldn't have shown up had I been using
          the operator-control TC (part of the communication model).
          [3 min.]

ME  1334: Bug #16:  Printing first text in loop multiple times.
    1335: Bug #16 fixed.  Left out :select-once-only.  Calling this a
          ME for the same reason as Bug #15.
          [1 min.]

ME  1336: Bug #17:  Gen-report selected multiple times.
    1337: Bug #17 fixed.  Proposal rules wasn't :select-once-only.
          Calling this a ME for the same reason as Bug #15.
          [1 min.]

    1340: continue test/debug
           Ok, the design seems to work.

    1340: begin coding

    1450/ short break
           Have all but last report chapter coded.

    1510: begin test/debug
```

```
TY 1511: Bug #18:  Extra parens around structured value spec.
   1512: Bug #18 fixed.  Wasn't using the structured editor for that TC.
         [1 min.]

TY 1512: Bug #19:  Premature end of file.
   1513: Bug #19 fixed.  I have an emacs function to help find these now,
         it is much easier.  I was missing the closing " on a text string.
         [1 min.]

ME 1516: Bug #20:  declarations detected bad attribute name.
   1517: Bug #20 fixed.
         [1 min.]

   1542: continue test/debug
          Long uninterrupted run to this point from 1517, after which I checked
          the generated report.

IE 1542: Bug #21:  Got a print loop wrong, begin/end itemize should be
         outside the loop.
   1548/ Bug #21 fixed, I think.  The test run will take a while.  Take
         a break while it runs.
         [6 min.]

   1615: resume test/debug
          Yes, bug #21 is really fixed.

   1615: begin coding
          Code the report generations for the last chapter.

   1645/ break

   1945: resume coding

   2029: begin test/debug

   2041: continue test/debug
          From 2029 to now was an unattended run.  At this point I started
          checking the generated report.

IE 2043: Bug #22:  Outside cable response sections missing.
   2048: Bug #22 probably fixed.  I had the same variable used in two
         different places in a TC where I should have had two different
         variables.
         [5 min.]

IE 2052: Bug #23:  Two sections for each inside cable, one with wrong
         enclosure name (it said the cable goes from an enclosure to
         itself).
   2054: Bug #23 fixed.  Missing a test to require that two values bound
         from a multi-attribute were different values.
         [2 min.]

IE 2055: Bug #24:  Equipments being reported as both hard and not hard.
   2057: Bug #24 fixed.  A copy-and-edit error.  Thinko.
         [2 min.]

   2057: continue test/debug
          I don't see any more problems.  Rerun from beginning to make
          sure.

   2102: continue test/debug
          From 2057 to now was an unattended run.  Check the report output.
```

```
2107/ initial task acquisition done
      The output is right.  Done!
```

Appendix E
Additional TAQL Experiment Data

This appendix presents the raw data from the TAQL experiments that does not appear in Tables 4-2 through 4-4 in Chapter 4. Tables E-1 through E-3 list the starting and ending dates for each task developed, the number of symbols in the source code for each task (*TC Symbols*), and the number of symbols in the compiled Soar productions for each task (*SP Symbols*). Section E.1 presents detailed information on each of the bugs subjects found in their programs, broken down by experiment round, subject, and task.

	Who	Task Name	Start Date	End Date	TC Symbols	SP Symbols
1	S1	MaC		4 Feb 1990	949	2882
2		KaW	6 Feb 1990	7 Feb 1990	959	3047
3		Hanoi	13 Feb 1990	13 Feb 1990	881	2595
4		Seq	19 Feb 1990	21 Feb 1990	2781	9589
Averages					**1392**	**4528**
5	S2	KaW	26 Jan 1990	26 Jan 1990	706	3173
6		BeaW	28 Jan 1990	1 Feb 1990	1951	7431
7		MaB	6 Feb 1990	6 Feb 1990	346	1710
8		Trav	6 Feb 1990	6 Feb 1990	773	3516
9		RaD	13 Feb 1990	19 Feb 1990	826	3882
10		IofQ	14 Feb 1990	14 Feb 1990	1617	6329
Averages					**1037**	**4340**
11	S3	MaB	31 Jan 1990	31 Jan 1990	987	3465
12		Truck	3 Feb 1990	4 Feb 1990	3894	11282
13		Birds	9 Feb 1990	9 Feb 1990	1232	6170
14		Sched	13 Feb 1990	13 Feb 1990	3528	11068
Averages					**2410**	**7996**
Overall avg.					**1531**	**5439**

Notes, by row:

13 Excludes tool-generated TCs (153 TCs/153 SPs)

Table E-1: Additional task development data: Round 1

	Who	Task Name	Start Date	End Date	TC Symbols	SP Symbols
1	S1	Sched	27 Sep 1990	2 Oct 1990	1898	6867
2		Truck	11 Oct 1990	19 Oct 1990	2893	15219
3		Mater	22 Oct 1990	22 Oct 1990	982	4998
4		Birds	23 Oct 1990	26 Oct 1990	777	2632
Averages					**1638**	**7429**
5	S2	Truck	9 Oct 1990	11 Oct 1990	2600	8458
6		Birds	13 Oct 1990	14 Oct 1990	3277	6215
7		Sched	14 Oct 1990	16 Oct 1990	1914	7480
8		Mater	18 Oct 1990	18 Oct 1990	4153	11330
Averages					**2986**	**8371**
9	S3	Mater	4 Oct 1990	5 Oct 1990	1190	4979
10		Pipe	15 Oct 1990	19 Oct 1990	6929	31945
11		Sizer	12 Nov 1990	26 Nov 1990	2764	8545
12		Pic	9 Dec 1990	9 Dec 1990	528	1765
13		Trav	10 Dec 1990	10 Dec 1990	663	2529
Averages					**2415**	**9953**
Overall avg.					**2351**	**8689**

Notes, by row:

3	Excludes tool-generated TCs (1 TC/59 SPs)
4	Excludes tool-generated TCs (121 TCs/121 SPs)
8	Excludes tool-generated TCs (93 TCs/93 SPs)
9	Excludes tool-generated TCs (40 TCs/40 SPs)

Table E-2: Additional task development data: Round 2

	Who	Task Name	Start Date	End Date	TC Symbols	SP Symbols
1	S3	Hanoi	29 May 1991	29 May 1991	465	1037
2		EMP	3 Jun 1991	9 Jun 1991	14313	34932
3		Predict	17 Jun 1991	23 Jun 1991	10531	17677
4		VT	25 Jun 1991	3 Jul 1991	20952	48333
Averages					**11565**	**25495**
Overall avg.					**11565**	**25495**

Notes, by row:

4 Excludes tool-generated TCs (72 TCs / 91 SPs)

Table E-3: Additional task development data: Round 3

E.1. Detailed bug data

During the TAQL experiments, subjects logged each bug they found in their implementations. They recorded when they detected a bug and the symptoms, and when and how they fixed it. After the experiments, the logs were examined to determine common classes of bugs. The following ten categories cover all of the bugs encountered in the experiments (each category is identified by a two-letter code):

- *TY: Typos.* A syntax error in the TAQL code. This does not include typos in user-defined items such as attribute names. Such typos are classified as *model errors*, defined below.
- *DE: Design errors.* Gross errors in design that show up during implementation, by leading to poor, difficult, incorrect or impossible implementations without some redesign. An example is forgetting to design a problem space to perform some aspect of the task.
- *IE: Implementation errors.* Errors where the programmer failed to correctly encode the problem space design they had in mind. An example is forgetting to include code to edit some aspect of the state that is specified in the system's operator-level design.
- *ME: Model errors.* Model errors are a subclass of implementation errors, and are defined in more detail in Section 4.3.2.
- *UE: Understanding errors.* These are errors stemming from the developer's misunderstanding of the task description. Subjects detected only three understanding errors during the experiments.
- *TP: TAQL problems.* There are bugs due to misunderstanding TAQL's semantics. Subjects detected only one bug of this class during the experiments.
- *PR: Procedural errors.* Procedural miscues interacting with Soar or TAQL. An example is forgetting to reload a modified TC during debugging. Subjects detected only seven procedural errors during the experiments.
- *CE: Chunking errors.* An incorrect chunk (for example, an overgeneral chunk). Subjects detected only three chunking errors during the experiments because learning was rarely turned on. I did not want the additional difficulties of getting a system to learn properly to confound the data.[14] See Section 3.1 for more details.
- *TE: Text errors.* Mistakes in the task description itself. Nine text errors were detected during the experiments. Text errors are not included in the data presented in this dissertation, and the time spent fixing them is not included in the development times I report. The reason is that the errors were fixed after the experiments and would not recur if the experiments were

[14]In one of the two tasks where learning was used (Truck.S1), the subject was using a Soar database package that requires learning. In the other (Sched.S1), the subject believed that learning was essential to a reasonably efficient implementation given the methods they chose.

repeated. Deducting the time to fix them will allow researchers using the corrected descriptions to validly compare their results to the results reported here.

- *SE: Soar errors*. Bugs in Soar itself. Eight Soar errors were detected during the experiments. Like text errors, Soar errors are not included in the data presented in this dissertation, and the time spent working around them is not included in the reported development times. The reason is that all of these bugs have been reported and, to the best of my knowledge, fixed. Therefore these errors would not recur if the experiments were repeated, and excluding the time spent fixing them will allow valid comparisons if the experiments are ever repeated with improved versions of TAQL. Furthermore, I wished to evaluate TAQL's quality, not the quality of underlying Soar implementation.

The remainder of this section presents information on all of the bugs detected during the experiments except for text errors and Soar errors. The information is broken down by experiment round, subject, and task. For each bug, I give the class code, fix time, and a one-line description. Many of the brief descriptions will be cryptic to those not very familiar with Soar and TAQL, but space constraints prevent more complete descriptions. Personal pronouns in the bug summaries refer to the subject who developed to the task, not to me.

E.1.1. Round 1 bugs

		Subject S1, Task MaC
IE	45	All the log says is 'operator implementation error'.
SE	185	Weird chunking bug in Soar 5.1.0.
ME	30	Careless mistake in a search control TC.
DE	75	Got into an infinite loop for some reason.

		Subject S1, Task KaW
ME	5	Attribute tie due to missing '&'.
DE	15	Needed to make several operators indifferent (hadn't thought of this in the original design).
IE	20	Subgoal initial state wrong (probably this had to do with needing a two-phase initialization — like a sliding operator.
ME	11	Forgot to put in goal-test-group.
ME	10	First operator needed best preference.
IE	20	Goal-test-group not working right (log is vague).
ME	5	Forgot to put in code to print the final answer.
SE	79	Overgeneral internal chunk.
IE	25	Final answer not getting printed (log is vague).

Subject S1, Task Hanoi		
ME	8	Attribute tie, forgot '&' preference.
ME	30	Attribute name mismatch.
DE	8	One operator was just a special case of another, so it was redundant.
ME	14	Operator tie resulting from overgeneral proposal conditions on one operator in an (intended) sequence.

Subject S1, Task Seq		
ME	8	Attribute tie.
IE	5	Bad propose-op conditions allowed multiple instantiations of the same operator.
ME	4	Misspelled operator name in prefer TC.
TP	15	Goal-test-group doesn't fire if the initial state is the goal.
ME	5	Attribute tie.
TY	5	Did the wrong math in (compute ...).
IE	8	Trying to do a sliding init-state TC.
IE	9	Omitted an edit clause in apply-op.
ME	9	Something wrong with compare TC --> operator tie (log is vague, may have been a typo in the TC).
ME	5	Forgot propose-space TC.
IE	8	Bad propose-op conditions of some sort (log is vague).
IE	5	Result-superstate not working (log is vague).
IE	13	Apply-operator conditions too general.
TY	5	Typo in above fix? (log is vague).
DE	30	Wanted stuff to be sticky but used augment TC.
DE	27	Infinite loop caused by augment TC that tested for absence of item, then added that item.

Subject S2, Task BeaW		
TY	5	Some typos (log is vague).
PR	5	Miscues using Soar IO.

Subject S2, Task Trav		
ME	15	15-min worth of 'silly errors with connectedness'.

Subject S2, Task RaD		
TY	3	Incorrect reuse of variable.
TY	4	Unlinked conditions.
ME	5	Incorrect reuse of variable: '^smaller <s>, <s> a state'.

Subject S2, Task IofQ		
TY	3	Extra parens (in :new for result-superstate).
TY	3	Extra '^' symbol: '^^'.
TY	3	Missing paren (closing).
TY	2	Linked to wrong context object.
TY	1	Unspecified.
DE	12	Didn't understand function of an operator.
ME	15	Forgot to code an operator.
IE	3	Missing condition in an operator proposal.
IE	1	Missing condition in an operator proposal.
ME	2	Missing :select-once-only.
IE	5	Forgot to emit an object in an apply-operator.
ME	5	Sloppiness in communication between spaces.
IE	1	Missing condition in an operator proposal.
IE	4	Missing condition in an operator proposal.
IE	10	Incorrect conditions in an operator proposal.
PR	4	Didn't reload a patched TC.
PR	5	Forgot to run soar.

Subject S3, Task Birds		
TY	1	Left out :op keyword when required.
TY	3	Duplicate TC names.
ME	1	Misspelling in a TC led to operator tie.
ME	4	Operator re-applied because proposal rules still satisfied.
UE	8	Realized some bird characteristics can have multiple values.
SE	10	Bug in Soar's accept function.
DE	4	Asking questions more than once, incorrectly keeping track of what has been asked.

Subject S3, Task Sched		
TY	2	Unbound vars in conjunctive negation.
TY	2	Duplicate TC names.
ME	12	Operator parameter id mismatch with copied object in lookahead.
IE	7	Forgot to update ^last-scheduled on object when operator applied.
ME	9	Search control not applying — misremembered data representation.
ME	8	Getting ties among best operators, which aren't resolved by the lookahead search (which makes best preferences).
IE	18	Sticky changes made on basis of goal test that later retracts.
IE	15	Flags not retracting because supported by copy productions.
IE	20	Made preferences for ops instead of eval-ops, which makes it impossible to backtrack when preferences yield a unique choice.
IE	5	Missing value after negated attribute.
ME	1	Missing parallel preferences --> attribute tie.

E.1.2. Round 2 bugs

Subject S1, Task Sched		
TY	4	Syntax errors in original code.
SE	51	Overgeneral internal chunk.
CE	10	Overgeneral real chunk.
CE	5	Still overgeneral real chunk.
DE	21	Misdesigned the whole search control mechanism.

Subject S1, Task Truck		
IE	5	Forgot :ind in operator control TC.
DE	8	Misdesigned failure mechanism — needed worst operator.
TY	5	Put in '>' instead of '>='.
TY	10	Various syntax errors caught by compiler.
ME	10	Attribute name mismatch.
ME	7	Class name mismatch.
ME	5	Operator no-change caused by attribute name error in apply-operator conditions.
PR	15	Copied database files in between machines without realizing the floating point formats would be different.
PR	10	Upper/lower case differences caused by *print-case* in Lisp reader/printer goofed up the database.
ME	5	Value name mismatch.
ME	6	Attribute name mismatch.
IE	5	Bad propose-op conditions (log is vague).
CE	30	Overgeneral chunk.

Subject S1, Task Mater		
ME	3	Operator name mismatch.
ME	2	Value name mismatch.
TY	5	Some typo (log is vague).

Subject S1, Task Birds		
DE	10	Tons of operators got made pairwise indifferent --> SLOW!.
IE	8	Propose-init-state didn't work the way I thought it did.
DE	4	Needed to make the final operator best; I hadn't realized any other operator could still be proposed.

Subject S2, Task Truck

TY	1	Extra '^' symbol.
TY	1	Missing paren.
ME	10	Mistyped 'leg' as 'let'.
ME	1	Missing parallel preferences on 4 attributes.
ME	10	Mistyped 'license' as 'licence'.
UE	10	Forgot a task constraint (white the criminal).
IE	1	Objects incorrectly structured in propose-initial-state. Thinko.
ME	10	Accessing the wrong state for information in propose-initial-state.
ME	5	Operator argument type mismatch (id vs. name).
ME	5	Incorrect operator parameter.
ME	20	Information not copied to local state.
IE	5	Missing test in operator proposal.
IE	5	Missing termination condition on operator.
IE	5	Conditions reversed on dual edit clauses. Thinko.
IE	5	Missing test in operator proposal.
DE	5	Forgot to represent some attributes.
IE	20	Forgot to generate a new attribute in propose-initial-state.
ME	5	Operator argument type mismatch (id vs. name).
ME	2	Forgot :select-once-only.
IE	2	Had math in an augment wrong. Thinko.
IE	5	Incorrect failure condition.
PR	5	Didn't reload a patched TC.

Subject S2, Task Birds

TY	1	Extra parens around a CE.
TY	10	Mistyped 'name' as '<name>'.
TY	1	Extra parens around a CE.
ME	1	Missing parallel preference.
TY	1	Missing paren.
IE	1	Attribute omitted in propose-task-state.
ME	1	Accessing local state when info not there.
ME	5	Didn't carry through an attribute name change.
IE	5	Missing test in operator proposal.
IE	5	Wrong (but extant) attribute in operator proposal.
IE	10	Wrong (but extant) attribute in operator proposal.
IE	10	Missing test in operator proposal.
IE	20	Combinatorics with comparative preferences.
IE	15	Combinatorics with comparative preferences.
DE	20	Incorrect behavior fixed with new code.
DE	10	Suboptimal behavior fixed with new code.
DE	10	Suboptimal behavior fixed by tweaking existing code.
PR	3	Remnants of old code left after a rewrite.

Subject S2, Task Sched

TY	1	Mistyped 'task-state' as 'top-state'.
TY	1	Mistyped 'lookahead' as 'lookhead'.
TY	7	Missing parens in argument to :new.
ME	20	Incorrect reuse of variable: '^triple <t> ... ^time <t>'.
ME	2	Incorrectly gave multiple values to an attribute.
IE	20	Incorrectly bound multiple objects in a :what clause. Thinko.
DE	120	Poorly formulated global search control.
DE	30	Local search control reversed (during design).

Subject S2, Task Mater

TY	1	Mistyped 'task-state' as 'top-state'.
TY	1	Extra paren.
ME	1	Mistyped attribute name.
ME	3	Mistyped attribute name.
ME	2	Mistyped attribute name.
ME	10	Forgot code for exiting from a space.
DE	5	Forgot to represent distance knowledge.
ME	5	Accessing local state when info not there.
ME	1	Used ^name attribute of operator in a proposal.
IE	10	Had search control coded wrong in operator-control. Thinko.
ME	1	Wrong attributes in conditions of an augment.
IE	10	Had search control coded wrong in operator-control. Thinko.
IE	1	Incorrect test in an augment.
TY	20	Repeat names in the editor-generated augments.
IE	1	Old code left over after editing.
IE	1	Old flag left over after editing.

Subject S3, Task Mater

TY	1	Left id variable out of an action.
TY	7	Duplicate TC name.

Subject S3, Task Pipe		
IE	1	Duplicate variable name led to attribute tie.
IE	4	Inconsistent volume units.
TY	6	Used wrong id variable to link cycles (typo).
SE	22	Reorder bug for autonomous productions.
TY	3	Missing id variable in condition. Misleading error message.
IE	8	Forgot goal test flag, causing space to not exit.
SE	16	Internal chunk bug.
DE	10	Keeps trying to fix soft constraint violations when it can only alleviate (not fix) the violations.
TY	19	Assorted syntax errors on load, most of the time spent finding a } typed for a), which Emacs's paren balancer didn't catch.
ME	5	Using wrong att name in proposal, not copying att from superspace.
ME	3	Using nil instead of false as attribute value.
TE	5	Sign error in formula in description.
IE	4	Multiple apply-operator TCs applying for op, due to incorrect copy/edit.
ME	5	Used goal instead of state in condition.
DE	3	Overgeneral goal test.
DE	5	Got formula wrong (incorrect derivation).
DE	43	Missed cases in a formula computation.
ME	5	Using wrong op name in apply-operator.
ME	4	Left out proposal TC for an operator.
IE	1	Operator applied to wrong pumping cycle.
IE	15	Compute-cycle-volume not stopping at end of cycle.

Subject S3, Task Sizer		
TY	7	Assorted syntax errors on load.
IE	2	Was not attaching object to the state, just accessing by :bind.
ME	5	Using wrong operator name.
ME	3	Testing negated attribute on wrong object.
IE	3	Basing computation on similar-case instead of customer-case.
ME	3	Misspelled variable name in expression.
IE	2	Didn't declare operator as optional in operator-control.

Subject S3, Task Pic		
UE	2	Made some wrong inferences from the task constraints.
IE	14	Left out :group-type in goal-test, causing novalue evaluations later (I wanted success). Hard to track down, since I didn't know about this feature, and I had rules that gave novalue.
TY	3	Used :type instead of :group-type.

Subject S3, Task Trav		
TY	1	Missing parens in :substructure value.
TY	1	Typed 'evaluate-operator' instead of 'evaluate-object' (TC name).
ME	2	State no-changes caused by mis-remembered class name.
ME	4	Overgeneral proposal TC, not testing that it hasn't applied.

E.1.3. Round 3 bugs

		Subject S3, Task Hanoi
DE	3	Overgeneral operator proposal, forgot about some conditions.
IE	1	Thinko, typed 'unknown' instead of 'novalue'.

		Subject S3, Task EMP
TY	1	Unlinked condition errors — bad variable names.
TY	4	Duplicate TC name.
TY	2	Premature EOF.
ME	1	Attribute on wrong object.
ME	1	Wrong attribute name.
ME	1	Missing application for operator.
ME	4	Wrong att name in (superoperator ...).
ME	1	Attribute tie — missing parallel preference.
SE	8	Soar transforms ^a (read) + & to ^a (read) + ^ (read) &.
DE	14	Couldn't handle list of zero responses to a question.
IE	3	Had code left over in an apply-operator TC from an outdated design.
IE	8	Coded one case of an operator wrong — a complex design.
TE	27	Sign error in formula in task description.
IE	1	Wrong (but legal) att name in a goal test.
ME	3	Unexpected operator tie, missing worst preference.
ME	1	Operator selected many times. Missing :select-once-only.
ME	1	Another operator selected many times. No :select-once-only.
TY	1	Extra parens around structured value spec.
TY	1	Premature EOF.
ME	1	Wrong (undeclared) attribute name.
IE	6	Put text inside a loop that should have been outside.
IE	5	Same var in two places in TC should be different vars.
IE	2	Missing test that two multi-att values are not equal.
IE	2	Didn't change enough in copy/edit for two similar TCs. Thinko.

Subject S3, Task Predict

TY	1	Left :value out of prefer.
IE	1	Using prefix instead of infix in compute.
ME	1	Forgot to make print-ict best.
IE	2	Tested extra stuff in goal test conditions.
SE	11	Soar was scrambling one of my productions.
TE	313	Parts of description massively wrong.
TY	1	Premature end-of-file.
ME	1	Undeclared attribute name.
IE	3	Forgot to compute value. Caught by way I used declarations.
IE	3	Forgot to compute value. Caught by way I used declarations.
IE	2	Forgot to compute value. Caught by way I used declarations.
IE	2	Forgot to compute value. Caught by way I used declarations.
IE	2	Forgot to compute value. Caught by way I used declarations.
IE	2	Forgot to compute value. Caught by way I used declarations.
IE	2	Forgot to compute value. Caught by way I used declarations.
IE	2	Forgot to compute value. Caught by way I used declarations.
IE	2	Forgot to compute value. Caught by way I used declarations.
IE	1	Got a formula wrong.
IE	8	Got conditions on a formula wrong.
IE	4	Got conditions on a formula wrong. Copy and edit error.
IE	1	Misspelled RHS var --> unbound --> bad value in working memory.
IE	2	Got a formula wrong.
TE	7	Test-case output in task description wrong.
TE	5	Test-case output in task description wrong.

Subject S3, Task VT

ME	1	Wrong operator name.
ME	1	Undeclared operator.
ME	1	Misspelled operator name.
ME	3	Undeclared attribute.
ME	1	Undeclared operator.
ME	1	Undeclared attribute.
ME	6	Missing declaration parts (attribute value types).
ME	1	Undeclared attribute.
ME	1	Undeclared attribute.
ME	1	Wrong type name in declaration.
ME	1	Undeclared attribute.
ME	1	Undeclared attribute.
IE	2	Operator not doing the right thing.
DE	52	Designed wrong semantics into a fix type.
ME	8	Same wrong att name in declaration and TC.
DE	9	Tried to access substructure that has retracted.
DE	10	Tried to access substructure that has retracted.
IE	26	Overconstrained augment — bad copy and edit.
DE	4	<= instead of < in comparison.
IE	1	Wrong TC type. Thinko.
ME	1	Missing declaration parts (attribute value types).
TY	1	Unbound variable passed to compute.
TY	3	Missing right paren, extra left paren later.
IE	1	Wrong TC type. Thinko.
TY	2	Missing right paren, extra left paren later.
TY	2	Missing right paren, extra left paren later.
TY	3	Duplicate TC name.
ME	2	Missing attribute declarations.
IE	1	Missing augment to compute formula.
ME	4	Misspelled attribute name.
IE	5	Missing augment to compute formula.
IE	4	Wrong numeric constant in condition.
IE	1	Underconstrained conditions in TC.
IE	1	Max-elaborations exceeded — VT might do this.
IE	4	Putting results on wrong state.
IE	3	Had two implementations of the same formula.
IE	3	Assigned wrong value to numeric constant.
IE	4	Switched top/bottom formulas for some loads. Thinko.
TE	51	Test case results in description wrong.
IE	3	Missing ready-to-fix rules for hand-coded fixes.
TE	58	Test case results in description wrong.
IE	11	Missing square root in formula.
DE	19	Confusion about O-support.
IE	4	Missing a case in a constraint.
DE	19	Trying to fix unfixable constraint repeatedly.
TE	19	Wrong constants in task description.
TE	102	Bad formulas in description.

References

Bachant, J. (1988). RIME: Preliminary work toward a knowledge-acquisition tool. In Marcus, S. (Ed.), *Automating Knowledge Acquisition for Expert Systems*. Boston, MA: Kluwer Academic Publishers.

Barker, V. E., O'Connor, D. E., Bachant, J., and Soloway, E. (March 1989). Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM, 32*(3), 298-317.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5: An introduction to rule-based programming*. Reading, Massachusetts: Addison-Wesley.

Chandrasekaran, B. (1986). Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design. *IEEE Expert, 1*, 23-30.

Clancey, W. (1983). The advantages of abstract control knowledge in expert system design. *Proceedings of the Third National Conference on Artificial Intelligence*. Washington, D.C..

Coleman, D., Holland, P., Kaden, N., Klema, V. and Peters, S. C. (1980). A system of subroutines for iteratively re-weighted least-squares computations. *ACM Trans. Math. Soft., 6*, 327-336.

Dhaliwal, J. S. and Benbasat, I. (1990). A framework for the comparative evaluation of knowledge acquisition tools and techniques. *Knowledge Acquisition, 2*, 145-166. Based on a paper presented at the 4th AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, October, 1989.

Doorenbos, B., Tambe, M. and Newell, A. (1992). Learning 10,000 chunks: what's it like out there? Computer Science Department, Carnegie Mellon University, January, 1992. Unpublished. To appear in the Proceedings of the 1992 National Conference on Artificial Intelligence, San Jose, CA.

Eshelman, L., Ehret, D., McDermott, J., and Tan, M. (1988). MOLE: A tenacious knowledge-acquisition tool. In Boose, J. and Gaines, B. (Eds.), *Knowledge Acquisition Tools for Expert Systems*. San Diego, CA: Academic Press.

Filman, R. E. (1988). The Big Giant Trucking Problem. Intellicorp, Inc. 1975 El Camino Real West, Mountain View, CA 94040. Unpublished.

Filman, R. E. (1988). Reasoning with worlds and truth maintenance in a knowledge-based programming environment. *Communications of the ACM, 31*(4), 382-401.

Forgy, C. L. (July 1981). *OPS5 User's Manual* (Tech. Rep. CMU-CS-81-135). Computer Science Department, Carnegie Mellon University.

Gaines, B. R. (1990). An architecture for integrated knowledge acquisition systems. *Proceedings of the 5th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Alberta, Canada.

Gruber, T. R. and Cohen, P. R. (1988). Design for acquisition: principles of knowledge-system design to facilitate knowledge acquisition. In Boose, J. and Gaines,

B. (Eds.), *Knowledge Acquisition Tools for Expert Systems.* San Diego, CA: Academic Press.

Hayward, S. A., Wielinga, B. J., and Breuker, J. A. (1988). Structured analysis of knowledge. In Boose, J. and Gaines, B. (Eds.), *Knowledge Acquisition Tools for Expert Systems.* San Diego, CA: Academic Press.

Johnson, T.R. (1991). *Generic Tasks in the Problem-Space Paradigm: Building Flexible Knowledge Systems While Using Task-Level Constraints.* Doctoral dissertation, Laboratory for Knowledge-Based Medical Systems, Ohio State University.

Johnson, T. R., Smith, J. W., Jr., and Chandrasekaran, B. (1989). Generic tasks and Soar. *Working Notes of the AAAI-89 Spring Symposium.* AAAI.

Johnson, K. A., Johnson, T. R., Smith, J. W., Jr., DeJongh, M., Fischer, O., Amra, N. K., and Bayazitoglu, A. (1991). RedSoar: A system for red blood cell antibody identification. *Proceedings of the Fifteenth Annual Symposium of Computer Applications in Medical Care.* Washington, D.C., McGraw Hill.

Johnston, Todd R., Smith, Jack W., Jr. and Chandrasekaran, B. . (1990). Task-Specific Architectures for Flexible Systems. Department of Computer and Information Science, Ohio State University, November, 1990, Unpublished.

Klinker, G. (1988). KNACK: Sample-driven knowledge acquisition for reporting systems. In Marcus, S. (Ed.), *Automating Knowledge Acquisition for Expert Systems.* Boston, MA: Kluwer Academic Publishers.

Klinker, G., Boyd, C., Dong, D., Maiman, J., McDermott, J. and Schnelbach, R. (1989). Building expert systems with KNACK. *Knowledge Acquisition, 1,* 299-320.

Klinker, G., Bhola, C., Dallemagne, G., Marques, D., and McDermott, J. (1990). Usable and reusable programming constructs. *Proceedings of the 5th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop.* Banff, Alberta, Canada.

Laird, J.E., Congdon, C.B., Altmann, E. and Swedlow, K. (October 1990). *Soar User's Manual: Version 5.2* (Tech. Rep.). Electrical Engineering and Computer Science, University of Michigan. Also available from The Soar Project, School of Computer Science, Carnegie Mellon University, CMU-CS-90-179.

Marcus, S. (1988). Taking backtracking with a grain of SALT. In Boose, J. and Gaines, B. (Eds.), *Knowledge Acquisition Tools for Expert Systems.* San Diego, CA: Academic Press.

Marcus, S., McDermott, J. and Wang, T. (August 1985). Knowledge acquisition for constructive systems. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.*

Marcus, S., Stout, J., and McDermott, J. (1988). VT: An expert elevator configurer that uses knowledge-based backtracking. *AI Magazine,* Vol. *9*(1).

McDermott, J. (1982). R1: A rule-based configurer of computer systems. *Artificial Intelligence, 19*(2), 39-88.

McDermott, J. (1986). Making expert systems explicit. *Proceedings of the Tenth Congress of the International Federation of Information Processing Societies.* Dublin, Ireland.